«Информационное тематическое пособие в помощь начинающему QA engineer (Тестировщику ПО)»

Автор и редактор:
QA engineer – Ильин А.Н.

2023 год

СОДЕРЖАНИЕ:

ТЕОРИЯ ТЕСТИРОВАНИЯ:

Расскажите о себе?

Почему вы решили стать тестировщиком?

(Пример: Потому что без тестирования невозможно выявить истинное состояние производимого продукта, и насколько он соответствует ожиданиям потребителя.)

Тестирование программного обеспечения — это проверка соответствия между реальным и ожидаемым поведением программы, а также выявление, насколько ПО удовлетворяет потребности пользователя и требованиям заказчика. Оно осуществляется на конечном наборе тестов, который составляет тестировщик.

Цель тестирования - проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы.

Для чего проводится тестирование ПО?

- Для проверки соответствия требованиям.
- Для обнаружения проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта.
- Обнаружение вариантов использования, которые не были предусмотрены при разработке. А также взгляд на продукт со стороны пользователя.
- Повышение лояльности к компании и продукту, т.к. любой обнаруженный дефект негативно влияет на доверие пользователей.

Этапы тестирования *(Жизненный цикл тестирования - совокупность выполнения всех этапов):*

- Инициация тестирования и Анализ продукта (событие, которое извещает команду тестирования о необходимости сессии тестирования, а также гарантирует выполнение требований к продукту для проведения тестирования).
- Выявление и анализ требований.
- Разработка стратегии тестирования и планирование процедур контроля качества.
- Создание тестовой документации (генерация и отбор тестовых случаев).
- Тестирование прототипа (процесс оценки первого проекта любого продукта).
- Основное тестирование.
- Создание отчётов о ходе и результатах тестирования (фиксация результатов).
- Стабилизация.
- Оценка качества объекта тестирования (анализ результатов).
- Эксплуатация.

Что можно и нужно тестировать:

- код (область модульного (unit) тестирования);
- software (софт, сам продукт) и hardware (взаимодействие софта с железом);
- prototype проекта (сырой продукт (может измениться):
- документация (требования, спецификация).

Жизненным циклом программного обеспечения (SLC) является период времени, начинающийся с момента появления концепции ПО и заканчивающийся тогда, когда использование ПО более невозможно. Жизненный цикл программного обеспечения обычно включает в себя следующие этапы: концепт, описание требований, дизайн, реализация, тестирование, инсталляция и наладка, эксплуатация и поддержка и, иногда, этап вывода из эксплуатации. Данные фазы могут накладываться друг на друга или проводиться итерационно.

Жизненным циклом разработки программного обеспечения (SDLC) является концепция, которая описывает комплекс мероприятий, выполняемых на каждом этапе (фазе) разработки программного обеспечения.

Этапы:

- принятие решение о необходимости создания продукта;
- сбор и анализ требований к проекту;
- проектирование (дизайн (Системы и ПО) на основе требований);
- реализация (кодирование на основе дизайна системы);
- тестирование продукта;
- внедрение и поддержка (сопровождение (в том числе фиксация найденных в пользовательской среде ошибок)).

Преимущество использования модели жизненного цикла разработки ПО (SDLC):

- обеспечение основы проекта (методологии, активность...);
- обеспечение визуализации хода реализации проекта;
- помощь компании в эффективности и успешного завершения проекта (сокращение затрат, уменьшение сроков разработки и тестирования, повышение качества конечного продукта);
- уменьшение рисков, связанных с процессом разработки ПО;
- обеспечение специальным механизмом отслеживания прогресса проекта.

Принципы тестирования:

1– Тестирование показывает наличие ошибок, а не их отсутствие.

Тестирование ПО сокращает количество ошибок. Оно снижает вероятность того, что необнаруженные ошибки останутся, но даже если ничего не было найдено, это не является доказательством исправности. Даже многократное тестирование никогда не может гарантировать, что программное обеспечение на 100% не содержит ошибок. Тестирование уменьшает их количество, но не устраняет.

2– Исчерпывающее тестирование невозможно.

Невозможно протестировать все функциональные возможности со всеми допустимыми и недопустимыми комбинациями данных во время фактического тестирования. Вместо этого подхода рассматривается тестирование нескольких приоритетных комбинаций с использованием различных методов.

Например, если у вас есть поле ввода, которое принимает буквы (имя), представьте, сколько имен будет проверяться – невозможно проверить все комбинации для каждого типа ввода.

3– Раннее тестирование.

Чтобы обнаружить ошибку в программном обеспечении, необходимо начать раннее тестирование. Ошибка, выявленная на ранних этапах жизненного цикла разработки ПО, обойдется гораздо дешевле. Для повышения качества программного обеспечения тестирование должно быть запущено на начальном этапе, т.е. выполняться на этапе анализа требований. Затраты, необходимые для устранения ошибки, обнаруженной в этот момент, меньше, и они продолжают расти по мере перехода к этапу тестирования или технического обслуживания.

4– Кластеризация дефектов.

Кластеризация дефектов означает, что небольшое количество модулей содержит в себе большинство обнаруженных ошибок. Это закон Парето, примененный к тестированию программного обеспечения: примерно 80% проблем, обнаруживаются в 20% модулей.

5– Тестирование зависит от контекста.

Подход к тестированию зависит от контекста разрабатываемого программного обеспечения. Различные типы тестирования должны выполняться для различных типов ПО. Например, тестирование сайта отличается от тестирования приложения для Android.

6– Парадокс пестицида.

Многократное повторение одних и тех же тестовых кейсов с одними и теми же тестовыми данными не приведет к обнаружению новых ошибок. Поэтому необходимо

проанализировать тестовые кейсы и обновить их или добавить другие, чтобы найти новые ошибки.

7- Заблуждение в отсутствии ошибок.

Если версия встроенного программного обеспечения на 99% рабочая, но не соответствует пользовательским запросам, то она непригодна для использования. Необходимо не только, чтобы программное обеспечение на 99% не содержало ошибок, оно также обязательно должно выполнять все требования пользователя. В таких случаях даже своевременные обнаружение и устранение ошибок не помогут, поскольку тестирование будет выполняться на основе неправильных требований, несоответствующих потребностям конечного пользователя.

QC (Quality Control) — **Контроль качества продукта** — это совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии в разрезах: **(задачи контроля качества)** «готовность продукта к выпуску», «соответствие зафиксированным требованиям», «соответствие заявленному уровню качества продукта».

QA (Quality Assurance) — **Обеспечение качества продукта** — это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации программного обеспечения (ПО) информационных систем, предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта, где тестирование является только одним из аспектов обеспечения качества.

К задачам обеспечения качества относятся:

- проверка технических характеристик и требований к ПО;
- оценка рисков;
- планирование задач для улучшения качества продукции;
- подготовка документации, тестового окружения и данных;
- тестирование;
- анализ результатов тестирования, а также составление отчетов и других документов.

Обеспечение качества определено в стандарте ISO 9000:2005 «Системы менеджмента качества. Основные положения и словарь» как «часть менеджмента качества, направленная на создание уверенности в том, что требования к качеству будут выполнены».

Верификация (verification) — это процесс оценки системы, чтобы понять, удовлетворяют ли результаты текущего этапа разработки условиям и требованиям, которые были сформулированы в его начале.

Валидация (validation) - это определение итогового результата, соответствия, разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе.

Существует шесть базовых типов задач:

Эпик (epic) — большая задача, на решение которой команде нужно несколько спринтов.

Требование (requirement) — задача, содержащая в себе описание реализации той или иной фичи.

История (story) — часть большой задачи (эпика), которую команда может решить за 1 спринт.

Задача (task) — техническая задача, которую делает один из членов команды.

Подзадача (sub-task) — часть истории / задачи, которая описывает минимальный объем работы члена команды.

Баг (bug) — задача, которая описывает ошибку в системе.

Тестовые среды:

Среда разработки (Development Env) – за данную среду отвечают разработчики, в ней они пишут код, проводят отладку, исправляют ошибки

Среда тестирования (Test Env) – среда, в которой работают тестировщики (проверяют функционал, проводят smoke и регрессионные тесты, воспроизводят.

Интеграционная среда (Integration Env) – среда, в которой проводят тестирование взаимодействующих друг с другом модулей, систем, продуктов.

Предпрод (Preprod Env) – среда, которая максимально приближена к продакшену. Здесь проводится заключительное тестирование функционала.

Продакшн среда (Production Env) – среда, в которой работают пользователи.

Основные фазы тестирования:

Pre-Alpha: прототип, в котором всё ещё присутствует много ошибок и наверняка неполный функционал. Необходим для ознакомления с будущими возможностями программ.

Alpha: является ранней версией программного продукта, тестирование которой проводится внутри фирмы-разработчика.

Beta: практически готовый продукт, который разработан в первую очередь для тестирования конечными пользователями.

Release Candidate (RC): возможные ошибки в каждой из фичей уже устранены и разработчики выпускают версию на которой проводится регрессионное тестирование.

Release: финальная версия программы, которая готова к использованию.

ТРЕБОВАНИЯ — это спецификация (описание) того, что должно быть реализовано. Требования описывают то, что необходимо реализовать, без детализации технической стороны решения.

Источники требований:

- заказчик;
- мозговой штурм;
- документы;
- фокус группа;
- наблюдение;
- моделирование;
- анкетирование;
- прототип;
- описание;
- нормы;
- лучшие практики;
- конкуренты.

Виды требований:

1) Прямые (формализованные в технической документации, спецификации, юзер-стори и прочих артефактах) и Косвенные (проистекающие из прямых, либо являющимися негласным стандартом для данной продукции или основывающихся на опыте и здравом смысле использования продукта); (пример про ссылку)

2) Функциональные (Уровни требований: бизнес-требования, требования пользователей, функциональные требования) и Нефункциональные (описание производительности, интерфейсы работы (платформа, протоколы)).

Атрибуты требований:

Корректность — точное описание разрабатываемого функционала.

Проверяемость — формулировка требований таким образом, чтобы можно было выставить однозначный вердикт, выполнено все в соответствии с требованиями или нет.

Полнота — в требовании должна содержаться вся необходимая для реализации функциональности информация.

Недвусмысленность — требование должно содержать однозначные формулировки.

Непротиворечивость — требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.

Приоритетность — у каждого требования должен быть приоритет (количественная оценка степени значимости требования). Этот атрибут позволит грамотно управлять ресурсами на проекте.

Атомарность — требование нельзя разбить на отдельные части без потери деталей.

Модифицируемость — в каждое требование можно внести изменение.

Прослеживаемость — каждое требование должно иметь уникальный идентификатор, по которому на него можно сослаться.

МОДЕЛИ РАЗРАБОТКИ:

1. Каскадная (водопадная (Waterfall)) базовым принципом является последовательный порядок выполнения задач. Это значит, что мы можем переходить к следующему шагу разработки или тестирования только после того, как предыдущий был успешно завершен. (**Анализ требований, проектирование, разработка, тестирование, техническая поддержка**).

2. V-Model (Модель верификации и валидации) основана на прямой последовательности шагов, тестирование в данном случае планируется параллельно с соответствующей стадией разработки. Согласно этой методологии тестирования ПО, процесс начинается, как только определены требования и становится возможным начать статическое тестирование, т.е. верификацию и обзор, что позволяет избежать возможных дефектов ПО на поздних стадиях. Соответствующий план тестирования создается для каждого уровня разработки ПО, что определяет ожидаемые результаты, а также критерии входа и выхода для данного продукта.

Основные этапы этой методологии могут изменяться, однако обычно они включают следующие:

Этап определения требований. Приемочное тестирование относится к этому этапу. Его основная задача состоит в оценке готовности системы к финальному использованию.

Этап, на котором происходит высокоуровневое проектирование, или High-Level Design (HDL). Этот этап относится к системному тестированию и включает оценку соблюдения требований к интегрированным системам.

Фаза детального дизайна (Detailed Design) параллельна фазе интеграционного тестирования, во время которой происходит проверка взаимодействий между различными компонентами системы.

После этапа написания кода начинается другой важный шаг — юнит-тестирование. Очень важно убедиться в том, что поведение отдельных частей и компонентов ПО корректно и соответствует требованиям.

3. Инкрементная модель:

Данная методология может быть описана, как мультикаскадная модель тестирования ПО. Рабочий процесс разделяется на некоторое количество циклов, каждый из которых также делится на модули. Каждая итерация добавляет определенный функционал к ПО. Инкремент состоит из трех циклов:

- дизайн и разработка;
- тестирование;
- реализация.

В этой модели возможна одновременная разработка разных версий продукта. Например, первая версия может проходить этап тестирования в то время, как вторая версия находится на стадии разработки. Третья версия в то же самое время может проходить этап дизайна. Этот процесс может продолжаться до самого завершения проекта.

4. Спиральная модель:

Спиральная модель — это методология тестирования ПО, которая основана на инкрементном подходе и прототипировании. Она состоит из четырех этапов:

- Планирование;
- Анализ рисков;
- Разработкам
- Оценка;

Сразу после того, как первый цикл завершен, начинается второй. Тестирование ПО начинается еще на этапе планирования и длится до стадии оценки. Основным преимуществом спиральное модели является то, что первые результаты тестирования появляется незамедлительно после появления результатов тестов на третьем этапе каждого цикла, что помогает гарантировать корректную оценку качества. Тем не менее, важно помнить о том, что эта модель может быть довольно затратной и не подходит для маленьких проектов.

5. Agile (SCRUM, CANBAN):

Методология гибкой (**Agile**) разработки и тестирование ПО может быть описана как набор подходов, ориентированных на использование интерактивной разработки, динамического формирования требований и обеспечения их осуществления как результата постоянного взаимодействия внутри самоорганизующейся рабочей группы. Большинство гибких методологий разработки ПО нацелены на минимизацию рисков посредством разработки в рамках коротких итераций. Одним из главных принципов этой гибкой стратегии является возможность быстрого реагирования на возможные изменения, нежели стремление положиться на долгосрочное планирование.

«SCRUM» - это процессный фреймворк, предназначенный для быстрой разработки и поставки сложных продуктов клиентам с максимально возможной ценностью

процесс разработки разбивается на отдельные этапы, результатом каждого из которых является готовый продукт. В конце каждого этапа (в терминологии Scrum — спринта) готовый продукт предоставляется заказчику. Полученный от заказчика отзыв позволяет выявить возможные проблемы или пересмотреть некоторые аспекты первоначального плана.

Прежде чем приступить к описанию жизненного цикла Scrum-проекта, стоит рассказать об ***основных ролях, принятых в Scrum-методологии***:

Владелец продукта (Product owner) представляет интересы конечного пользователя.

Скрам-мастер (Scrum master) следит за соблюдением принципов Scrum-разработки, координирует процесс, проводит ежедневные собрания (Scrum Meetings).

Скрам-команда (Scrum team) участвует в разработке продукта. В скрам-команду входят программисты, тестировщики, аналитики и прочие специалисты. (5-9 чел.)

Стейкхолдеры (бизнес-заказчик) и пользователи.

Шаг 1. Создание бэклога продукта

Бэклог продукта (Product backlog) представляет собой упорядоченный по степени важности список требований, предъявляемых к разрабатываемому продукту. Элементы этого списка называются Пользовательскими историями (User story (описание функциональной возможности ПО простыми словами, составленное с точки зрения пользователя)). Каждой истории соответствует уникальный ID.

Описание каждой истории должно включать в себя набор обязательных полей, необходимых для дальнейшей работы над проектом:

Важность (Importance). Степень важности задачи, по мнению владельца продукта. Описывается произвольным числом.

Предварительная оценка (Initial estimate). Предварительная оценка объема работ. Измеряется в story point’ах.

Как продемонстрировать (How to demo). Описание способа демонстрации завершенной задачи.

Шаг 2. Планирование спринта и создание Бэклога спринта

На этапе планирования определяется длительность спринта. Короткий спринт позволяет чаще выпускать работающие версии продукта, а, следовательно, чаще получать отзывы от клиента и вовремя выявлять возможные ошибки. С другой стороны, длинные спринты позволяют посвятить решению проблемы больше времени. Оптимальная длина спринта выбирается как нечто среднее между этими двумя решениями и составляет обычно около 2-ух недель. На этом этапе важно взаимодействие владельца продукта и скрам-команды. Владелец продукта определяет приоритет той или иной задачи, а задача скрам-команды состоит в определении необходимых трудозатрат.

Во время планирования спринта команда выбирает самые приоритетные пользовательские истории из бэклога продукта и решает, каким образом будут решаться поставленные задачи. Истории, выбранные для реализации в течение данного спринта, составляют Бэклог спринта (Sprint backlog). Количество историй, попадающих в бэклог спринта зависит от их длительности в story point'ах, присвоенных каждой истории на этапе предварительной оценки. Это количество выбирается так, чтобы каждая история была успешно реализована к концу спринта.

Шаг 3. Работа над спринтом. Scrum meetings

После того, как определены актуальные для данного спринта пользовательские истории, начинается процесс разработки.

Для визуализации процесса разработки удобно использовать учетные карточки. Они могут иметь вид больших карточек с названием конкретной истории и маленьких стикеров, описывающих отдельные задачи, необходимые для реализации истории. После начала работы над определенной задачей, ее стикер перемещается из поля «Запланировано» в область «В работе». По завершении работы над задачей, стикер перемещается в поле «Тестирование» и затем, при успешном выполнении тестирования, в поле «Готово». Расположив истории согласно их важности, можно получить представление о текущем состоянии проекта:

Также может быть использовано программное обеспечение, предназначенное для такого рода задач. Примером такого ПО может служить, например, Atlassian JIRA.

Важной отличительной особенностью Scrum являются ежедневные совещания (Scrum meetings), целью которых является дать команде полную и достоверную информацию о том, на каком этапе находится процесс разработки. Во время совещания каждый участник скрам-команды сообщает о том, какая задача им выполнена, какая будет выполняться и какие у него возникли трудности во время работы.

Шаг 4. Тестирование и демонстрация продукта

Поскольку в идеале результатом каждого спринта является продукт, готовый к работе, важное место в Scrum занимает процесс тестирования. Существуют разные способы свести к минимуму затраты на данном этапе: от уменьшения количества историй в спринте и, как результат, снижения количества ошибок до включения тестировщиков в скрам-команду.

Финал каждого спринта — демонстрация готового продукта. Скрам-команда составляет ревью, в котором описывает цели спринта, поставленные задачи и то, как они были решены. Владелец продукта, заказчики и пользователи на основе ревью и демонстрации принимают решение о том, что должно быть изменено в дальнейшем процессе разработки.

Шаг 5. Ретроспектива. Планирование следующего спринта

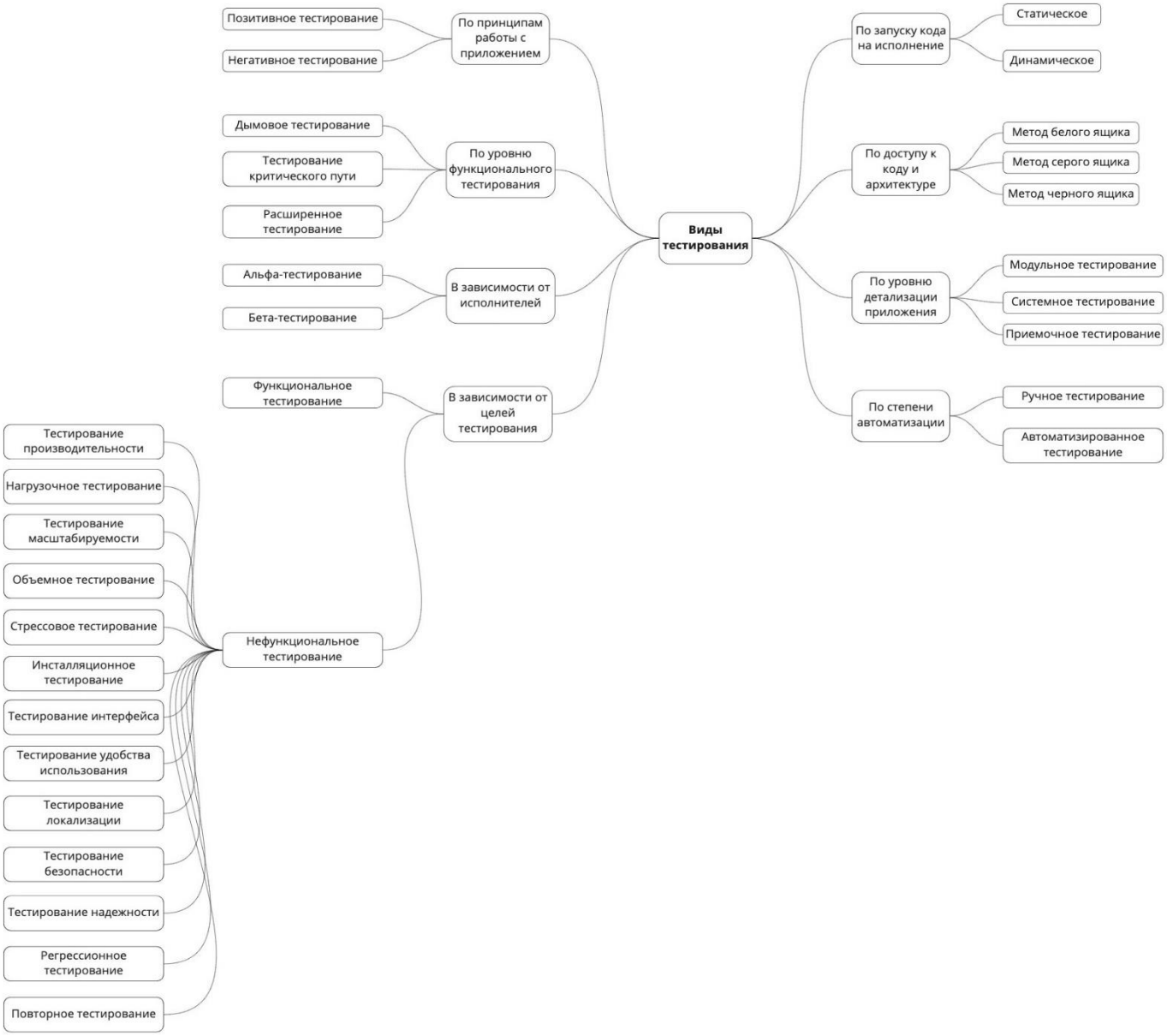
На основе отзыва о продукте, полученного после демонстрации, проводится ретроспектива. Ее основная цель — определить, как можно улучшить процесс разработки на следующем спринте, чтобы избежать возникших проблем и работать более эффективно. После того, как пути улучшения качества работы были определены, команда может приступать к планированию следующего спринта.

«KANBAN» стал символом визуализации рабочего процесса (карточки на стенде) и сейчас обозначает стратегию оптимизации потока поставки ценности посредством

процесса, использующую систему вытягивания и имеющую ограничение незавершённой работы. Этот метод, демонстрирующий, что происходит в процессе работы, увеличивает предсказуемость процедур, благодаря чему рабочий процесс становится прозрачным и равномерным. Kanban также можно использовать, чтобы достичь большей согласованности действий, а это означает более быстрое достижение стратегических целей.

ВИДЫ ТЕСТИРОВАНИЯ:

Вид тестирования — это совокупность активностей, направленных на тестирование заданных характеристик системы или её части, основанная на конкретных целях.



1) Классификация по запуску кода на исполнение:

Статическое тестирование — процесс тестирования (без запуска кода), который проводится для верификации практически любого артефакта разработки: программного кода, требований, системных спецификаций, функциональных спецификаций, документов проектирования, архитектуры программных систем и их компонентов. (Предотвращение деф.)

Динамическое тестирование — тестирование проводится на работающей системе, не может быть осуществлено без запуска программного кода приложения. (Поиск и испр. деф.)

2) Классификация по доступу к коду и архитектуре (*Методы тестирования** (на сколько глубоко можно погрузиться в техническую составляющую продукта, к потоку хранения информации и ее передачи, к кодовой базе)**):

Тестирование белого ящика — метод тестирования ПО, который предполагает полный доступ к коду (внутренняя структура/устройство/реализация системы) проекта.

Согласно ISTQB, тестирование белого ящика — это:

тестирование, основанное на анализе внутренней структуры компонента или системы;

тест-дизайн, основанный на технике белого ящика — процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.

Тестирование серого ящика — метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично (DevTools, база данных, API, техническая документация, логи).

Тестирование чёрного ящика — метод тестирования ПО, который не предполагает доступа (полного или частичного) к системе. Основывается на работе исключительно с внешним интерфейсом тестируемой системы.

Согласно ISTQB, тестирование черного ящика — это:

тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы;

тест-дизайн, основанный на технике черного ящика — процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

3) Классификация по уровню детализации приложения:

Модульное тестирование (Unit test) — проводится для тестирования какого-либо одного логически выделенного и изолированного элемента (модуля) системы в коде. Проводится самими разработчиками, так как предполагает полный доступ к коду.

Интеграционное тестирование — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.

Системное тестирование — процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности. (Набор end-to-end tests).

Приёмочное тестирование — проверяет соответствие системы потребностям, требованиям и бизнес-процессам пользователя.

4) Классификация по степени автоматизации:

Ручное тестирование.

Автоматизированное тестирование.

5) Классификация по принципам работы с приложением:

Позитивное тестирование — тестирование, при котором используются только корректные (валидные) данные.

Негативное тестирование — тестирование приложения, при котором используются некорректные (не валидные) данные и выполняются некорректные операции.

6) Классификация по уровню функционального тестирования:

Дымовое тестирование (smoke test) — тестирование, выполняемое на новой сборке, с целью подтверждения того, что программное обеспечение стартует и выполняет основные для бизнеса функции. (***Связанные с изменениями***:

Санитарное тестирование (Sanity testing) - проверка того, что определённые части приложения так же работают как положено после минорных изменений или исправлений багов.

Регрессионное тестирование (regression testing) — тестирование уже проверенной ранее функциональности после внесения изменений в код приложения, для уверенности в том, что эти изменения не внесли ошибки в областях, которые не подверглись изменениям.

Повторное/подтверждающее тестирование (re-testing/confirmation testing) — тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок).

Тестирование критического пути (critical path) — направлено для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности.

Расширенное тестирование (extended) — направлено на исследование всей заявленной в требованиях функциональности.

7) Классификация в зависимости от исполнителей:

Альфа-тестирование — является ранней версией программного продукта. Может выполняться внутри организации-разработчика с возможным частичным привлечением конечных пользователей.

Бета-тестирование — программное обеспечение, выпускаемое для ограниченного количества пользователей. Главная цель — получить отзывы клиентов о продукте и внести соответствующие изменения.

8) Классификация в зависимости от целей тестирования:

Функциональное тестирование (functional testing) — направлено на проверку корректности работы функциональности приложения. (Подразумевает что нужно проверить все функции программы и сравнить фактический результат с ожидаемым).

Этапы функционального тестирования:

- требования (спецификации, user-story, примеры конкурентов и др.);
- входные данные (позитивные и негативные сценарии);
- выходные данные (ожидаемый результат);
- прохождение сценариев (получение фактического результата);
- сравнение результатов.

Нефункциональное тестирование (non-functional testing) — тестирование атрибутов компонента или системы, не относящихся к функциональности.

Тестирование производительности (performance testing) — определение стабильности и потребления ресурсов в условиях различных сценариев использования и нагрузок.

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

Тестирование масштабируемости (scalability testing) — тестирование, которое измеряет производительность сети или системы, когда количество пользовательских запросов увеличивается или уменьшается.

Объёмное тестирование (volume testing) — проводится для тестирования программного приложения с определенным объемом данных.

Стрессовое тестирование (stress testing) — проверка, как система обращается с нарастающей нагрузкой (количеством одновременных пользователей).

Тестирование на отказ и восстановление (помехоустойчивость) – проверка способности противостоять и успешно восстанавливаться после сбоев, в связи с ошибками ПО, отказами оборудования или проблемами связи/сети.

Инсталляционное тестирование (installation testing) — тестирование, направленное на проверку успешной установки и настройки, обновления или удаления приложения.

Тестирование интерфейса (GUI/UI testing) — проверка требований к пользовательскому интерфейсу.

Тестирование удобства использования (usability testing/user experience UX) — это метод тестирования, направленный на установление степени удобства использования, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Конфигурационное тестирование — это проверка работы программного обеспечения на различных программных и аппаратных окружениях. Данный вид тестирования применяется, если известно, что информационный продукт будет использоваться с разной конфигурацией, на разных платформах, в различных браузерах, будет поддерживать разные версии драйверов.

Тестирование совместимости - предназначено для определения того, может ли программное обеспечение или продукт работать в различных браузерах, базах данных, оборудовании, операционной системе, мобильных устройствах и сетях.

Тестирование локализации (localization testing) — проверка адаптации программного обеспечения для определенной аудитории в соответствии с ее культурными особенностями. (*Интернациализации* – как расположен текст в окне).

Тестирование безопасности (security 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 надёжности (reliability testing) - проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

Тестирование документации (Documentation testing) - минимизации рисков несоответствия фактически реализованной функциональности и прохождения приемочных испытаний.

АРТЕФАКТЫ ТЕСТИРОВАНИЯ:

Внешняя документация:

- ***замечание*** (короткая записка, комментарий о небольшой неточности в реализации продукта);
- ***баг-репорт;***
- ***запрос на изменение (улучшение)*** (описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им.)
- ***отчет о тестировании (тест репорт)*** (документ, предоставляющий сведения о соответствии/несоответствии продукта требованиям).

Внутренняя документация:

- ***тест-план;***
- ***тестовый сценарий*** (последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразным им проверок корректности поведения продукта в ходе этих действий);
- ***чек-лист***
- ***тест-кейс;***
- ***тест-сьют.***

Проектная документация — включает в себя всё, что относится к проекту в целом.

Продуктовая документация — часть проектной документации, выделяемая отдельно, которая относится непосредственно к разрабатываемому приложению или системе.

Тест план (Test Plan) — это документ, который описывает 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. (Является артефактом тестирования)

Тест-планы бывают:

1) Приемо-сдаточный план (набор смоук-тестов);
2) Сам тест-план (динамичный (или версии плана), почти готовый отчет):

- что надо тестировать;
- что будем тестировать;
- как будем тестировать;
- когда будем тестировать;
- критерии начала тестирования;
- критерии сдачи и приемки.

Основные пункты тест плана: (в идеале)

- Идентификатор тест плана (Test plan identifier);
- Введение (Introduction);
- Объект тестирования (Test items);
- Функции, которые будут протестированы (Features to be tested;)
- Функции, которые не будут протестированы (Features not to be tested);
- Тестовые подходы (Approach);
- Критерии прохождения тестирования (Item pass/fail criteria);
- Критерии приостановления и возобновления тестирования (Suspension criteria and resumption requirements);
- Результаты тестирования (Test deliverables);
- Задачи тестирования (Testing tasks);
- Ресурсы системы (Environmental needs);
- Обязанности (Responsibilities);
- Роли и ответственность (Staffing and training needs);
- Расписание (Schedule);
- Оценка рисков (Risks and contingencies);
- Согласования (Approvals).

Пункты плана: (в жизни)

- Перечень работ;
- Критерии качества и оценки (ошибки на релизе и др.);
- Оценки рисков (области в которых не квалифицированы, время, деньги на закупку устройств, замена тестировщика и др.);
- Документация (тест-кейсы или чек-листы, шаблоны тестовых атрибутов, как часто тест-кейсы должны пересматриваться, будет ли ревью кейсов внутри команды, сроки приемки тест-кейсов);
- Тестовая стратегия (методы, уровни, виды тестирования, виды для конкретного модуля, объем регрессионного тестирования, метрики по передаче тест-кейсов в регресс, сроки принятия исправленных дефектов);
- Ресурсы (человеческие (сколько и кого), аппаратные (ноуты, телефоны), программные (лицензии программ, платные интерфейсы и софт));
- Расписание (даты, контрольные точки).

3) Мастер план (требования к заводимым дефектам, условия принятия сборки, критерии принятия продукта к релизу, инструменты).

Чек-лист (Check list) — список проверок (идея проверки), без указания шагов и ожидаемого результата. (Группируются по смыслу, модулям) (Для небольших проектов)

Атрибуты чек-листа (не исчерпывающий перечень, гибкость составления):

- Номер;
- Описание (название, идея) проверки;
- Статус (результат);
- Комментарий.

Плюсы: гибкость, простота создания и поддержки, простота визуализации, расширение тестового покрытия, каждый выполняет по-своему.

Минусы: каждый выполняет по-своему, неопределенность тестовых данных, неэффективен для джунов, высокая вероятность разночтения.

Основные программы: Excel, Trello, Jira + плагины.

Тестовый сценарий (Test-case) — это артефакт, описывающий совокупность шагов, конкретных условий (пред- и пост-), параметров (входных значений) и ожидаемых результатов, необходимых для проверки реализации тестируемой функции или её части.

Атрибуты тест-кейса:

- *Номер кейса;*
- *Заголовок* (название);
- *Предусловия (PreConditions)* — список действий или условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
- *Шаги (Steps)* — список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям.
- *Ожидаемый результат (Expected result)* — что по факту должны получить.
- *Окружение*;
- *Тестовые данные*;
- *Статус;*
- *Фактический результат;*
- *Постусловия*;
- *Модуль;*
- *Приоритет;*
- *Уровень* (модульное, интеграционное, системное, приемочное);
- *Вид тестирования;*
- *Связанное требование;*
- *Комментарии.*

Правила написания тест-кейса:

1) независимость (тест-кейсов друг от друга);
2) однозначность (введите логин (чем?) латиницей);
3) полнота (никто не должен что-то где-то додумывать или спрашивать);
4) обезличенность (нажать на кнопку);
5) упрощай (НЕ НАДО - нажмите на красную кнопку с надписью: «Войти» в верхнем правом углу экрана, под меню; НАДО – нажать на кнопку «Войти»);
6) 1 тест-кейс = 1 проверка, цель (отдельно авторизация и покупка товара).

Классификация тест-кейсов: высокоуровневые (описывают поведение как в чек-листе, больше идея для проверки, чем подробное описание (нехватка времени) (больше для API)) и низкоуровневые (подробные).

Статусы проверки:

- Passed (пройден, успешен);
- Failed (сломался);
- Skipped (пропущен (нет необходимости, убрали функционал));
- No run (не запускался (по умолчанию));
- Blocked (заблокирован (каким-то найденным багом в предыдущем тесте)).

Отчёт о дефекте (bug report) — документ, который содержит отчет о любом недостатке в компоненте или системе, который потенциально может привести компонент или систему к невозможности выполнить требуемую функцию. (Описывает ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата). (Баг-трекеры: Jira, Redmine, Mantis, «Excel»).

Ошибка (Error) – несоответствие производимого продукта предъявляемым к нему прямым и косвенным требованиям. (Ошибка *пользователя*, то есть он пытается использовать программу иным способом (например, вводит буквы в поля, где требуется вводить цифры). В качественной программе предусмотрены такие ситуации и выдаются сообщение об ошибке (error message)). (Это ошибка *программиста (или дизайнера* или ещё

кого, кто принимает участие в разработке), то есть когда в программе, что-то идёт не так, как планировалось. Например, внутри программа построена так, что изначально не соответствует тому, что от неё ожидается).

Дефект, баг (defect, bug) — отклонение фактического результата от ожидаемого. (Недостаток в рабочем продукте, если он не соответствует его требованиям).

Failure (неудача) — это *сбой* в работе компонента, всей программы или системы (может быть, как аппаратным, так и вызванным дефектом).

Локализация дефекта:

– Выявление причины возникновения дефекта. (Например, не проходит восстановление пароля. Необходимо выявить, откуда приходит запрос на сервер в неверном формате – от backend или frontend).

- Исследовать окружение. (Необходимо воспроизвести баг в разных операционных системах (IOS, Android, Windows и т.д.) и браузерах (Google Chrome, Mozilla, Internet Explorer и др.). При этом нужно проверить требования к продукту, чтобы выяснить, какие системы должны поддерживаться.)
- Проверить на разных устройствах.
- Проверить на разных версиях ПО (версионность продукта).
- Проанализировать возможность влияния найденного дефекта на другие области.

Атрибуты отчета о дефекте:

1. Уникальный идентификатор (ID) — присваивается автоматически системой при создании баг-репорта.
2. Тема (краткое описание, Summary) — кратко сформулированный смысл дефекта, отвечающий на вопросы: Что? Где? Когда (при каких условиях)?
3. Подробное описание (Description) — более широкое описание дефекта (указывается опционально).
4. Шаги для воспроизведения (Steps To Reproduce) — описание четкой последовательности действий, которая привела к выявлению дефекта. В шагах воспроизведения должен быть описан каждый шаг, вплоть до конкретных вводимых значений, если они играют роль в воспроизведении дефекта.
5. Фактический результат (Actual result) — описывается поведение системы на момент обнаружения дефекта в ней. чаще всего, содержит краткое описание некорректного поведения (может совпадать с темой отчета о дефекте).
6. Ожидаемый результат (Expected result) — описание того, как именно должна работать система в соответствии с документацией.
7. Окружение (Environment) – окружение, на котором воспроизвелся баг.
8. Критичность (серьёзность) дефекта (важность, Severity) — характеризует влияние дефекта на работоспособность приложения.
9. Приоритет дефекта (срочность, Priority) — указывает на очерёдность выполнения задачи или устранения дефекта.
10. Статус (Status) — определяет текущее состояние дефекта. Статусы дефектов могут быть разными в разных баг-трекинговых системах.
11. Вложения (Attachments) — скриншоты, видео или лог-файлы.
12. Возможность обойти баг, воспроизводимость, комментарии (не обязательные атрибуты).

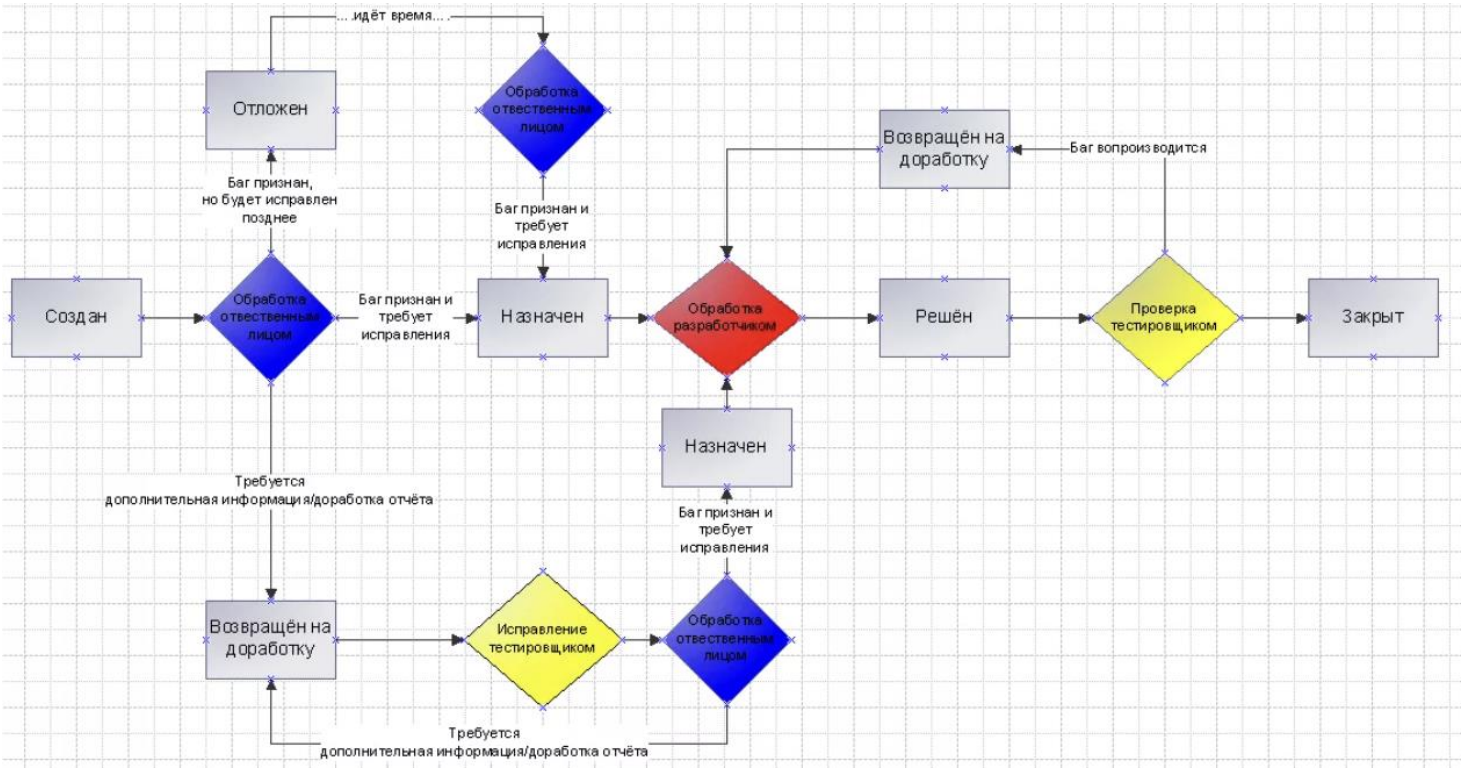
Шаги воспроизведения:

- описывайте каждое действие в отдельном шаге;
- описывайте безличные формулировки, призывающие к действию;
- описывайте каждый шаг, пока не столкнетесь с дефектом;
- найдите кратчайший путь воспроизведения;
- найдите точный путь воспроизведения;
- пишите так, чтобы любой новичок мог его воспроизвести.

Релиз багов – это преднамеренное действие, а **утечка багов** - случайное. ***Релиз багов*** подразумевает, что при отправке приложения команде тестировщиков разработчики

знали, что оно содержит ошибки. Но они могут быть не критичными, поэтому можно проводить релиз. ***Утечка багов*** подразумевает, что группа тестировщиков не выявила ошибку, и конечный пользователь или заказчик получает приложение с ошибкой. (Выяснить причину, провести анализ пропуска дефекта, внедрить корректирующие мероприятия, чтобы этого больше не происходило (Обсудить ситуацию с командой разработки, установить новые правила в команде, которые позволят повлиять на эту ситуацию).).

Жизненный цикл бага:



Severity vs Priority:

Критичность (серьёзность) (severity) показывает степень ущерба, который наносится проекту существованием дефекта. Severity выставляется тестировщиком.

Градация Критичности (серьезности) дефекта (Severity):

- **Блокирующий (S1 – Blocker)** тестирование значительной части функциональности вообще недоступно. Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
- **Критический (S2 – 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, то есть не работает важная часть одной какой-либо функции либо не работает значительная часть, но имеется workaround (обходной путь/другие входные точки), позволяющий продолжить тестирование.
- **Значительный (S3 – Major)** не работает важная часть одной какой-либо функции/бизнес-логики, но при выполнении специфических условий, либо есть workaround, позволяющий продолжить ее тестирование либо не работает не очень значительная часть какой-либо функции. Также относится к дефектам с высокими visibility – обычно не сильно влияющие на функциональность дефекты дизайна, которые, однако, сразу бросаются в глаза.
- **Незначительный (S4 – Minor)** часто ошибки GUI, которые не влияют на функциональность, но портят юзабилити или внешний вид. Также незначительные функциональные дефекты, либо которые воспроизводятся на определенном устройстве.

- **Тривиальный (S5 – Trivial)** почти всегда дефекты на GUI — опечатки в тексте, несоответствие шрифта и оттенка и т.п., либо плохо воспроизводимая ошибка, не касающаяся бизнес-логики, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Приоритет (срочность) (priority) показывает, как быстро дефект должен быть устранён. Priority выставляется менеджером, тимлидом или заказчиком.

Градация Приоритета (срочности) дефекта (Priority):

- **P1 Высокий (High)** Критическая для проекта ошибка. Должна быть исправлена как можно быстрее.
- **P2 Средний (Medium (Normal))** Не критичная для проекта ошибка, однако требует обязательного решения.
- **P3 Низкий (Low)** Наличие данной ошибки не является критичным и не требует срочного решения. Может быть исправлена, когда у команды появится время на ее устранение.

Высокий приоритет и низкая серьезность:

Такое сочетание бывает, когда баг на функционал влияет незначительно, но зато на пользовательский опыт влияет очень сильно. Также в эту категорию попадают баги, не влияющие на программу, но требующие исправления.

Вот пара примеров:

1. Кнопки перекрывают друг друга. Они кликабельны, но визуальное впечатление портится.
2. Логотип компании на главной странице содержит орфографическую ошибку. На функционал это вообще не влияет, но портит пользовательский опыт. Этот баг нужно исправить с высоким приоритетом, несмотря не то, что на продукт он влияет минимально.

Высокая серьезность и низкий приоритет:

Такое сочетание бывает у багов, которые возникают в отдельных функциях программы. Эти баги не позволяют пользоваться системой, при этом обойти их невозможно. Но сами функции, содержащие эти дефекты, конечным потребителем используются редко.

Примеры:

1. Домашняя страница сайта ужасно выглядит в старых браузерах. Перекрывается текст, не загружается логотип. Это мешает пользоваться продуктом, поэтому серьезность бага высокая. Но так как очень мало пользователей открывают сайт при помощи устаревшего браузера, такой баг получает низкий приоритет.
2. Допустим, у нас есть приложение для банкинга. Оно правильно рассчитывает ежедневный, ежемесячный и ежеквартальный отчет, но при расчете годового возникают проблемы. Этот баг имеет высокую степень серьезности. Но если сейчас формирование годовой отчетности не актуально, такой дефект имеет низкий приоритет: его можно исправить в следующем релизе.

Советы:

- убирайте лишние шаги;
- упрощайте шаги, чем длиннее шаг/описание шага, тем хуже;
- проверьте на дубликаты;
- убедитесь, что репорт очевидный и понятный;
- 1 отчет для 1 дефекта;
- прослеживаемость с другими артефактами (возможность привязаться к тест-кейсу, тест-сьюту, тестовому прогону, к пунктам требований);
- проверьте грамотность.

Что делать, если разработчик утверждает, что найденный дефект таковым не является?

- Указывать на требования, апеллировать к здравому смыслу, подключать аналитика, чтобы объяснил.

- Если это поведение в требованиях не указано, то обратиться к аналитику, за уточнением этого поведения.
- Если они считают, что все хорошо – дефект закрывается.

Основные ошибки:

- недостаточно предоставленных данных, для воспроизведения бага, или баг воспроизводится только при обстоятельствах, но они не указаны;
- название репорта и ожидаемый результат не соответствуют друг другу;
- завышение/занижение severity (критичности (серьезности) влияния на систему);
- неверное употребление терминологии;
- сложные речевые обороты;
- отсутствует ожидаемый результат;
- критика программиста.

Отчет о тестировании – часть тестовой документации, включающая в себя описание процесса тестирования, суммарную информацию о протестированных за подотчетный период билдах, информацию о деятельности тестировщиков, а также другие статистические данные (анализ результатов тестирования, который проводится с некоторой периодичностью (календарный график, например, раз в 2 неделю, в конце тестирования билда или в конце спринта) в процессе работы с проектом, а также в конце работы с проектом). Его ***основная задача*** – оценить текущее или финальное качество проекта и принять (если необходимо) соответствующие решения и меры. (Отчет создается на основе принятого в компании или предоставленного заказчиком шаблона).

Цель отчета – предоставить лицам, заинтересованным в проекте, полную и объективную информацию о текущем состоянии качества проекта. Эта информация выражается в конкретных фактах и цифрах.

Структура отчета:

- Титульный лист (название продукта, билд, версия, автор).
- Команда тестировщиков (задействованные лица с указанием должности и роли на проекте в подотчетный период).
- Описание процесса тестирования (testing process description) (краткое описание того, как происходило тестирование: окружение, какие использовались методы, техники, виды, инструментальные средства и т.п.).
- Краткое описание (summary) (какие билды были протестированы (прошел или не прошел тестирование), есть ли в качестве продукта прогресс или регресс, есть ли какие-либо проблемы, требующие внимания руководства).
- Расписание (testing timetable) (детализированное описание того, какая работа и на протяжении какого времени выполнялась каждым тестировщиком).
- Рекомендации (recommendations) (выделить важные моменты, на которые следует обратить внимание руководству или лидерам проектных команд, рекомендация если проект готов на передачу заказчику или в продакшн).
- Статистика по ошибкам (bags statistics). (Найдено, исправлено, проверено, отклонено, открыто заново (все этапы (статусы) с отображением критичности)).
- Список новых ошибок (new bugs found) (найденных за подотчетный период).
- Статистика по всем ошибкам (за все время работы над проектом) (all bugs statistics). (Найдено, исправлено, проверено, отклонено, открыто заново (все этапы (статусы) с отображением критичности)).

ТЕХНИКИ ТЕСТ-ДИЗАЙНА:

Тест-дизайн — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы). (Принцип Парето - «20% усилий дают 80% результата, а остальные 80% усилий — лишь 20% результата»).

Задачами тест-дизайна являются:

- Анализ требований и рисков тестирования;
- Определение проверок для тестирования;
- Формализация проверок в виде тестовых сценариев;
- Приоритезация проверок;
- Определение подходов к тестированию.

Техники тест-дизайна:

1. Тестирование на основе классов эквивалентности (equivalence partitioning) — это техника, основанная на методе чёрного ящика, при которой мы разделяем функционал (часто диапазон возможных вводимых значений, поле ввода) на группы эквивалентных по своему влиянию на систему значений, или которые приводят систему к одному результату.

То есть, это некое множество значений, которое вы можете подставлять в программу и получать один и тот же результат. Результатом можем быть не только конкретные значения, действия программы, но и просто область применения. Поэтому, самые простые классы эквивалентности, на которые делятся проверки, это 2 основных класса: позитивные (валидные значения) и негативные (не валидные значения) сценарии. Далее, каждый класс эквивалентности можем разделить на дополнительные классы и т.д. до того момента, пока проверки не будут приводить к точечным и конкретным результатам тестирования.

Рассмотрим пример:

Система скорринга рассчитывает процентную ставку по кредиту для клиента исходя из его возраста, который вводится в форму:

- От 18 до 25 лет – 18%
- От 25 до 45 лет – 16 %
- Свыше 45 лет – 20%

Мы определяем 2 основных класса – это **позитивные и негативные** сценарии.

Позитивными сценариями будут все значения, которые приводят к получению результата, негативными сценариями – значения, результаты которых не описаны, как ожидаемый результат.

Далее мы делим класс позитивных сценариев 3 класса вводимых значений 18-24, 25-44 и 45 +.

В классе негативных сценариев мы формируем значения, исходя из необходимости проверки отказов программы, поэтому мы имеем 0, 1-17, отрицательные значения, ввод символов и т.д.

Результатом данного разбиения будет значение или диапазон значений, в котором нам необходимо выполнить всего одну проверку с любым значением из диапазона данных. Могут возникнуть такие ситуации, как одно значение в диапазоне. Это тоже отдельный класс эквивалентности и тоже требует проверки.

Итого мы имеем.

- Позитивные проверки: Ввод значений: 19, 30, 48 (значения могут быть любыми из данного диапазона класса).
- Негативные проверки: 0, 3, -1, А и т.д.

2. Техника анализа граничных значений (boundary value testing) — это техника дополняет классы эквивалентности дополнительными проверками поведения продукта на крайних (граничных) значениях входных данных.

Граничные значения определяются не только для числовых значений, но и для буквенных (например, границы алфавита и кодировки), даты и времени, смысловых значений. Граница числовых значений зависит от формата ввода, если у вас целые числа, например, 2, то граничные значения будут 1 и 3. Если дробные значения, то границы для числа 2 уже будут 1,9 (1,99) или 2,1 (2,01) и т.д. Главное определить шаг.

Дополним вышеописанный пример:

Далее исключаем повторяющиеся значения, и получаем значения для проверки элемента ввода данных.

-1, 0, 1, 17, 18, 19, 24, 25, 26, 44, 45, 46, max.

Значение max обычно уточняется у Заказчика или аналитика. Если не могут предоставить, то нужно подобрать значение, соответствующее здравому смыслу (вряд ли кто-то придет за кредитом в возрасте 100 лет).

Следующий шаг, это наложить граничные значения на значения классов эквивалентности, исключить лишние проверки, пользуясь правилом «достаточно одного значения для проверки одного класса» и финализировать список.

3. **Попарное тестирование (pairwise testing)** — это одна из техник тест-дизайна, основанная на комбинаторике и разделению входных параметров «по парам» (почему и называется pairwise testing). Проводится комбинирование вариантов и подбор нужных, то есть оцениваются все возможные комбинации (сочетания) входных переменных, и из них выбираются только нужные (значимые). Техника основана на том, что 99,9…% дефектов возникают при взаимодействии не более двух факторов одновременно/

ISTQB определяет попарное тестирование как технику тест-дизайна методом черного ящика, при которой тест-кейсы создаются таким образом, чтобы выполнить все возможные отдельные комбинации каждой пары входных параметров.

Быстрый пример:

QA-команде передали приложение, в которое пользователь должен вводить свои значения. Всего 10 полей ввода, может быть по 10 вариантов в каждом. Получается всего 10x10=100 комбинаций нужно будет протестировать, если пойти по ошибочному пути «исчерпывающего тестирования».

Или, например, есть простое приложение, в которое ввод подается выбором значений из таких объектов ввода: выпадающего списка (с числами от 0 до 9), чекбокса, радиокнопки, текстового поля, и кнопки ОК. Текстовое поле принимает только числа от 0 до 100. Получаются варианты следующие:

Список: 0,1,2,3,4,5,6,7,8,9

Чекбокс: Отмечен / нет

Радиокнопка: Выбрана / не выбрана

Текстовое поле: числа 0…100

Сколько же тест-кейсов нужно создать? Невероятно много. *Исчерпывающее тестирование* такого приложения предполагает (умножаем все количества вариантов) 10*2*2*100 = 4000 тест-кейсов. Если же к этому добавить «негативные варианты» (то есть с вводом заведомо некорректных значений, а так обязательно случается в реальном мире), то количество будет «сильно выше» 4000.

Как же сократить это количество, упрощая себе задачу? То есть, в чем заключается суть попарного тестирования?

Во-первых, что здесь можно сократить? Нельзя — варианты радиокнопки и чекбокса, они всегда будут иметь только возможных 2 значения. С текстовым полем можно ограничиться тремя числами: валидное целое, невалидное целое, буква или спецсимвол. Со списком: здесь предположим, для упрощения, что значение будет или 0, или “любое другое кроме 0”, то есть получается только 2 варианта, «0» и «другое». Считаем количество вариантов (тест-кейсов) теперь: 2*2*2*3 = всего 24. То есть, при «обычном» подходе у нас 24 тест-кейса, что уже неплохо.

Идем дальше по пути сокращения:

1. Упорядочиваем объекты ввода так, чтобы объект ввода с самым большим количеством вариантов шел первым, и т.п.
2. Делаем таблицу. Список у нас будет принимать 2 значения.
3. Следующая колонка — чекбокс, тоже 2 значения.
4. Проверяем, что у нас «покрыты» все комбинации списка и чекбокса
5. То же делаем с радиокнопкой (2 значения)
6. Проверяем, что все пары «покрыты». (используем программу)

Текстовое поле	Выпадающий список	Чекбокс	Радиокнопка
Валидное	0	Отмечен	Включена
Валидное	“Другое значение” (не 0)	Не отмечен	Выключена
Невалидное	“Другое значение” (не 0)	Отмечен	Включена
Невалидное	0	Отмечен	Выключена
Невалидное	0	Не отмечен	Включена
Буква	0	Не отмечен	Включена
Буква	0	Отмечен	Выключена
Буква	“Другое значение” (не 0)	Отмечен	Включена

Таким образом, пользуясь техникой попарного тестирования, сократили количество тест-кейсов сначала с 4000 до 24, затем до 8 как в таблице, что уже вполне посильно. И при этом надежность такого метода вполне нормальная.

Полезности метода:

- Уменьшает количество тест-кейсов;
- Позволяет быстро повысить тестовое покрытие;
- Достаточно эффективно повышает % найденных багов;
- Ускоряет создание и выполнение тест-кейсов;
- Снижает издержки проекта.

Недостатки:

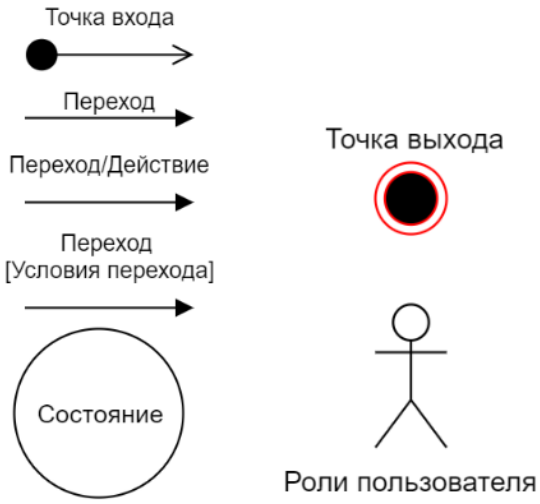
- Не работает с некоторыми типами переменных;
- Нужен опыт в определении комбинаций;
- Трудоемкость.

4. Тестирование на основе состояний и переходов (State-Transition Testing) — применяется для фиксирования требований и описания дизайна приложения. (Схема состояний и переходов (от англ. State & Transition Diagram, S&T) — это схема переходов и состояния, специальная техника для перехода ТЗ из одного статуса в другой. С ее помощью пользователь в наглядной форме может просматривать переход продукта из одной стадии в другую. Идеально подходит для длительных проектов, где техническое задание разбито на большие спринты, где требуется контроль и верификация любого действия). (Можно сказать, что это совокупность причин и следствий (но есть действия, которые не имеют определенного состояния)).

В проекте может быть большой набор требований с описанием состояния системы и условий, при которых она в них переходит. Без визуального представления этих состояний трудно увидеть всю цепочку событий. А это может привести к дефектам архитектуры и дизайна приложения уже на уровне требований. Например, теперь в мессенджерах можно удалять сообщения как у отправителя, так и у получателя. То есть для состояния сообщения «Отправлено» или «Прочитано» должен быть предусмотрен переход в состояние «Удалено». Если он будет упущен при составлении требований — приложение получится неудобным для пользователей, вряд ли его станут часто запускать.

Чтобы избежать таких ошибок, можно использовать технику тест-дизайна «Тестирование состояний и переходов». Она позволяет составлять тестовые сценарии, основываясь на визуальном представлении состояний и переходов системы.

Прежде чем рассматривать эту технику, познакомимся с основными понятиями, которые используются при составлении диаграмм переходов и состояний.



Точка входа — старт работы системы или приложения.

Переход (transition) — переход системы из одного состояния в другое, который происходит в результате действий пользователя или при определённых условиях.

Событие (event) — (надпись над стрелкой) — действие пользователя, которые он выполнил для перевода системы в другое состояние, или действия самой системы, меняющие её состояние.

Действие (action) — реакция приложения на действия пользователя или самой системы (на событие).

Условия перехода (transition conditions) — условия, которые необходимы для перехода системы в другое состояние, например, изменение даты для начисления процентов на вклад.

Состояние (state) — состояние системы до или после перехода в результате действий пользователя или при определённых условиях.

Точка выхода — успешное окончание полного цикла работы приложения, то есть выполнение всех переходов и состояний.

Роли пользователей (actors) — пользователи, которые могут по-разному влиять на систему в зависимости от уровня прав доступа (зарегистрированный пользователь, менеджер, администратор).

Классический пример — бронирование авиабилетов. Начнём с позитивного сценария: пользователь успешно осуществляет весь цикл бронирования, включая оплату и использование билета. Всегда стоит начинать с позитивных проверок, чтобы убедиться, что система работоспособна и выполняет ключевые функции. Если это не так, то дальнейшее тестирование не имеет смысла до устранения дефектов.

1. Точкой старта в этом случае будет вход в систему бронирования и выбор нужного билета. Затем пользователь передаёт информацию, необходимую для

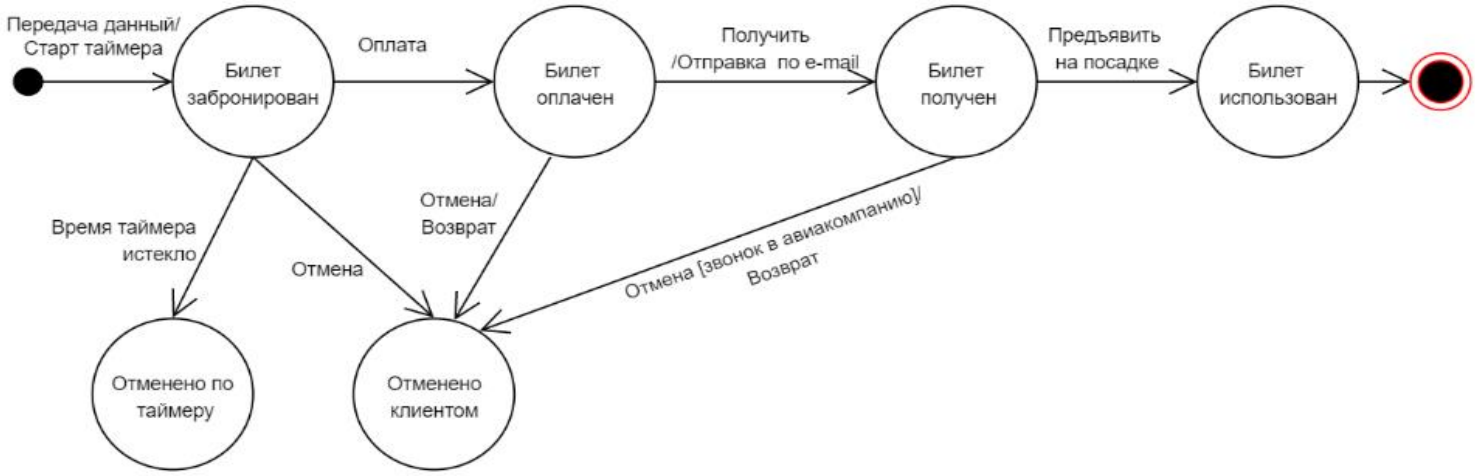
бронирования билета (имя и фамилию, паспортные данные), и нажимает кнопку «Забронировать». Это нажатие можно считать событием. Под его воздействием стартовал таймер времени до окончания срока оплаты. Система перешла в первое состояние «Билет забронирован».

2. Дальнейшее событие — «Оплата билета», которое переведёт систему в следующее состояние «Билет оплачен».

3. Затем по событию «Получить билет» система должна выполнить действие «Отправка билета по email» и перейти в другое состояние — «Билет получен».

4. Последним звеном в этой цепочке будет событие «Предъявление билета при посадке» (в примере часть событий пропущена — в реальных проектах их, конечно, может быть намного больше). Состояние «Билет использован» — цикл бронирования успешно завершён, система попадает в точку выхода.

Рассмотренный сценарий — позитивный, он не предполагает дополнительных действий пользователя. Это, конечно, невозможно, так как не всегда бронирование билета должно заканчиваться его использованием. Пользователь может не оплатить билет — или оплатить, но потом отменить, и прочее. Эти состояния также необходимо отразить на диаграмме.



Так как пользователь может забронировать билет, но не оплатить его, отмена брони произойдёт по истечении времени на оплату. В таком случае необходимо добавить в диаграмму состояние «Отменено по таймеру». А также «Отменено клиентом», в которое система может перейти из трёх предыдущих: «Билет забронирован», «Билет оплачен», «Билет получен».

При переходе из «Билет забронирован» в «Отменено клиентом» пользователю достаточно просто произвести событие «Отмена». Если билет уже был оплачен, действием системы должен стать возврат денежных средств — «Возврат».

Предположим, что для перехода системы из состояния «Билет получен» в «Отменено клиентом» помимо отмены билета пользователь должен выполнить условие перехода — позвонить в авиакомпанию. Это тоже необходимо отразить на диаграмме. Условие перехода указывается над стрелкой в квадратных скобках.

Так визуализируется работа системы бронирования, и можно сразу увидеть состояния, которые принимает система, и условия для их изменения. Такая визуализация актуальна для сложных проектов с множеством состояний, переходов и условий для них. Она позволяет не пропустить важные звенья системы и наиболее полно описать тестовые сценарии для проверки. Например, первым сценарием может стать проверка полного цикла работы системы от входа в неё до точки выхода. Затем уже можно выполнять тестирование более детально, добавляя новые сценарии на основании диаграммы переходов и состояний.

При построении диаграмм состояний и переходов стоит учитывать следующее:

- не допускать пересечения линий переходов — это усложняет визуальное восприятие диаграммы и может привести к ошибочному переходу;
- сложные процессы лучше представлять в виде нескольких диаграмм — если охватить всё одной схемой, она будет слишком сложной;
- главную последовательность состояний следует размещать на одной горизонтальной линии, чтобы прослеживался позитивный сценарий работы системы. Дополнительные состояния можно представить в виде ответвлений и разместить по бокам от основной последовательности.

Плюсы диаграмм состояний и переходов:

- позволяют визуализировать состояния продукта;
- демонстрируют варианты переходов, которые можно пропустить;
- помогают отследить дефект, сужая его локацию до конкретного перехода;
- показывают внутреннюю механику продукта.

Минусы:

- можно пропустить неочевидные переходы;
- при слишком сложной структуре продукта диаграммы могут стать громоздкими и запутанными;
- являются только основой к применению других методов;
- бесполезны при плохом знании продукта.

5. **Таблицы принятия решений (Decision Table Testing)** — техника тестирования, основанная на методе чёрного ящика, которая применяется для систем со сложной логикой, помогающая наглядно изобразить комбинаторику условий из ТЗ, повысить общее тестовое покрытие, не упуская все (возможные) комбинации.

Ячейки таблицы заполняются с опорой на три параметра, которые расположены в шапке и первом столбце. Всё начинается с условий работы системы, выбранных из требований. Далее идут правила, которые отражают выполнение условий. Завершается таблица действиями — это результаты, которые наступают при соблюдении правил.

	Правило 1	**Правило 2**	**...**	**Правило N**
Условия				
Условие 1				
Условие 1				
...				
Условие N				
Действия				
Действие 1				
Действие 2				
...				
Действие N				

Преимущества:

Наглядность. Это главное преимущество метода. Вместо того, чтобы текстом описывать тест-кейсы и бояться что-то упустить, можно составить матрицу и быть уверенным, что ни одна проверка не потеряется.

Удобство. Один столбец таблицы — один готовый тест-кейс.

Простота. Таблицу решений можно составить в Google-таблице, Excel, на бумаге или даже на салфетке, если хочется. Чтобы использовать метод, не нужно уметь писать код или осваивать специальную программу.

Недостатки:

Долго. Главный минус метода: для составления таблицы нужно время, которого всегда не хватает. Иногда тестировщик думает, что проще потратить полчаса-час на тестирование, а не на составление таблицы.

Рассмотрим составление таблицы на примере:

Требование: для поддержания системы лояльности провести информационную рассылку постоянным клиентам.

Содержание писем зависит от следующих условий:

1. Клиенты типа A, B получают стандартное письмо.
2. Клиенты типа C получают специальное письмо.

Клиентам, совершившим пять и более покупок или купившим на сумму более 500 долларов, в письме сообщается о дополнительной скидке 20% на следующий заказ.

Начнём составлять таблицу по плану:

1. Разбить требование на условия.
2. Посчитать количество возможных правил (комбинаций).
3. Составить таблицу принятия решений.
4. Исключить лишние комбинации, если они есть.
5. Создать тесты.

Разберём каждый пункт.

1. Разбить требование на условия.

В данном случае можно выделить три условия:

- тип клиента;
- пять и более покупок;
- сумма больше 500 долларов.

2. Посчитать количество возможных правил (комбинаций).

Расчёт можно выполнить по формуле: $X = Y1 \cdot Y2 \cdot \ldots \cdot Yn$, где:

- X — вычисляемое количество комбинаций;
- Y1...Yn — количество вариантов каждого условия;
- N — количество условий.

Таким образом получим:

- Y1 = 4 (четыре значения для условия «Тип клиента» — «A, B, C, D»);
- Y2 = 2 и Y3 = 2 (по два значения для условий «Пять и более покупок» и «Сумма больше 500 долларов» — «YES/NO»);
- N = 3 (требование содержит три условия);
- $X = 4 \cdot 2 \cdot 2$;
- X = 16 правил (комбинаций условий).

3. Составить таблицу принятия решений.

Заносим в таблицу условия, значения и правила следующим образом:

Условия	Значения	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Тип клиента	A ,B, C, D	A	A	A	A	B	B	B	B	C	C	C	C	D	D	D	D
5 и более покупок	Y, N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Сумма больше 500 дол	Y, N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Действия																	
Стандартное письмо		X	X	X	X	X	X	X	X					?	?	?	?
Специальное письмо										X	X	X	X	?	?	?	?
Сообщение о скидке		?	X	X		?	X	X		?	X	X		?	?	?	?
Не получают письмо														?	?	?	?

В таблицу был добавлен тип клиента «D» — это все остальные типы клиентов (если существуют), если будут выявлены те, что не подпадают под характеристики для клиентов типа «A, B, C». Для правил, которые не отражены в требованиях, использован «?» (в требованиях не указано, какое письмо должно быть отправлено в ситуации, когда сочетаются условия «более пяти покупок» и «сумма больше 500 долларов», а также как поступить с клиентами типа D). Ситуации, помеченные знаком вопроса, надо прояснить с аналитиком или заказчиком.

Первая строка в таблице формируется так: количество всех правил (комбинаций) делится на количество значений первого условия. То есть 16 (число правил) делим на 4 (число значений для условия «Тип клиента»). Получаем ряд из четырёх одинаковых значений подряд (см. таблицу выше). Заполняя остальные строки, необходимо соблюдать последовательность: каждая следующая строка — это предыдущая строка, разделённая пополам. То есть в первой строке каждое значение повторялось четыре раза подряд, во второй — два раза, в третьей уже происходит чередование значений. Если бы в таблице было ещё одно условие, то в следующей строке каждое значение снова повторялось бы четыре раза, потом два раза и так далее.

Также в таблице указываем действия, которые произойдут при совпадении тех или иных условий. И отмечаем, какое именно действие выполняется при совпадении условий. В данном случае выделим четыре действия: «Стандартное письмо», «Специальное письмо», «Сообщение о скидке», «Не получают письмо».

4. Исключить/добавить комбинации.

В этом случае были добавлены комбинации для дополнительного типа клиента «D». Также могут возникать ситуации, когда в таблице появятся комбинации условий, которые на практике невозможны. Например, если тестируется форма с двумя кнопками «Сохранить» и «Отменить», каждая кнопка имеет два значения: «Нажата» / «Не нажата». Одновременно обе кнопки не могут принимать значение «Нажата» — значит, такая комбинация должна быть исключена из таблицы.

5. Создать тесты.

В результате получаем тестовые сценарии, которые можно либо перенести в тест-кейсы, либо оставить в таблице и добавить строку с результатом проверки.

6. Доменный анализ (Domain Analysis Testing) — это техника основана на разбиении диапазона возможных значений переменной на поддиапазоны, с последующим выбором одного или нескольких значений из каждого домена для тестирования. (Используется для определения действенных и эффективных тестовых сценариев в случаях, когда множественные параметры могут или должны быть протестированы одновременно). (Комбинация техник классов эквивалентности, граничных значений, попарного тестирования).

Плюсы и минусы доменного тестирования:

Плюсы:

- Обнаружение ошибок при минимальном количестве тестов.
- Интуитивно понятный, универсальный подход.

Минусы:

- Низкая вероятность обнаружения ошибок НЕ на граничных условиях.
- Низкая вероятность обнаружения ошибок в сложных взаимодействиях.
- Пространство значений часто бывает сложно формализовать.

Пример техники доменного тестирования:

Представим ситуацию, когда нужно одновременно протестировать несколько параметров, для которых существуют линейные классы эквивалентности.

Сервис прогноза погоды развивается и обогащается функциональностью для профессиональных путешественников.

User story 1: Я как пользователь хочу узнать прогноз погоды, указав координаты точки на карте.

User story 2: Я как пользователь хочу узнать прогноз погоды на выбранное количество дней.

Пользователь: заполняет поле “Широта” значением от -90,000000 до 90,000000.

Пользователь: заполняет поле “Долгота” значением от -180,000000 до 180,000000.

Пользователь: заполняет поле “Дней” значением от 1 до 3.

Пользователь: выбирает язык.

Пользователь: выбирает информацию по осадкам.

Пользователь: выбирает детализацию по дням / часам.

Данные валидные:

Система: показывает прогноз погоды.

Данные невалидные:

Система: показывает сообщение об ошибке “Прогноз не найден. Уточните параметры поиска”.

Сколько нужно тестов, чтобы обеспечить оптимальное покрытие с учетом тестирования граничных значений?

- Поле “Широта”: 6 тестов
  - -90,000000
  - -90,000001
  - -89,999999
  - 90,000000
  - 90,000001
  - 89,999999
- Поле “Долгота”: 6 тестов
  - -180,000000
  - -180,000001
  - -179,999999
  - 180,000000
  - 180,000001
  - 179,999999
- Поле “Дней”: 5 тестов
  - 0
  - 1
  - 2
  - 3
  - 4
- Язык, осадки, детализация по pairwise: 4 теста – по технике “Классы эквивалентности”.

Итого 21 тест, если проверять все по отдельности. Но количество тестов можно сократить при помощи техники доменного анализа.

Основной принцип доменного анализа - скомбинировать значения на границах и внутри интервалов и таким образом сократить количество тест-кейсов. Доменный анализ оперирует понятиями:

- точка on - лежит строго на границе;
- точка off - лежит слева или справа от границы, т.е. точки on;
  - если интервал **закрыт** со стороны точки on, то точка off лежит **вне** интервала;
  - если интервал **открыт** со стороны точки on, то точка off лежит **внутри** интервала;
- точка in - любое значение внутри интервала, ближе к середине.

Алгоритм доменного анализа:

1. Создадим таблицу и внесем в нее:

a. параметры, для которых есть линейные классы эквивалентности;

b. для каждого параметра - граничные значения со знаками >, <, >=, <=;

c. для каждой границы - строки on, off;

d. для каждого параметра - значение in.

21	Широта	>= -90,000000	on
22			off
23		<= 90,000000	on
24			off
25			in
26	Долгота	>= -180,000000	on
27			off
28		<= 180,000000	on
29			off
30			in
31	Дней	>= 1	on
32			off
33		<= 3	on
34			off
35			in

2. Заполняем только строки on и off для всех параметров по диагонали (т.е. в одной колонке должно быть только 1 значение on или off для 1 параметра).

				Номер теста											
Параметр	Границы		1	2	3	4	5	6	7	8	9	10	11	12
Широта	>= -90,000000	on	-90											
		off		-90,000001										
	<= 90,000000	on			90									
		off				90,000001								
		in												
Долгота	>= -180,000000	on					-180							
		off						-180,00001						
	<= 180,000000	on							180					
		off								180,000001				
		in												
Дней	>= 1	on									1			
		off										0		
	<= 3	on											3	
		off												4
		in												

3. Теперь заполняем значения in. В каждой колонке в итоге должно быть значение on / off для 1 параметра и значение in для остальных.

			Номер теста											
Параметр	Границы		1	2	3	4	5	6	7	8	9	10	11	12
Широта	>= -90,000000	on	-90											
		off		-90,000001										
	<= 90,000000	on			90									
		off				90,000001								
		in					-65	-65	-65	-65	-65	-65	-65	-65
Долгота	>= -180,000000	on					-180							
		off						-180,00001						
	<= 180,000000	on							180					
		off								180,000001				
		in	135	135	135	135					135	135	135	135
Дней	>= 1	on									1			
		off										0		
	<= 3	on											3	
		off												4
		in	2	2	2	2	2	2	2	2				

4. Дополним таблицу оставшимися параметрами, которые были предварительно скомбинированы по принципу pairwise.

			Номер теста											
Параметр	Границы		1	2	3	4	5	6	7	8	9	10	11	12
Широта	>= -90,000000	on	-90											
		off		-90,000001										
	<= 90,000000	on			90									
		off				90,000001								
		in					-65	-65	-65	-65	-65	-65	-65	-65
Долгота	>= -180,000000	on					-180							
		off						-180,00001						
	<= 180,000000	on							180					
		off								180,000001				
		in	135	135	135	135					135	135	135	135
Дней	>= 1	on									1			
		off										0		
	<= 3	on											3	
		off												4
		in	2	2	2	2	2	2	2	2				
Язык			RU	US	RU	US	RU	RU	US	RU	US	RU	RU	US
Осадки			no	yes	yes	no	no	no	yes	yes	no	no	no	yes
Детализация			days	hours	days	days	hours	days	hours	days	days	hours	days	hours

5. Выделим красным цветом невалидные значения и добавим ожидаемый результат.

			Номер теста											
Параметр	Границы		1	2	3	4	5	6	7	8	9	10	11	12
Широта	>= -90,000000	on	-90											
		off		-90,000001										
	<= 90,000000	on			90									
		off				90,000001								
		in					-65	-65	-65	-65	-65	-65	-65	-65
Долгота	>= -180,000000	on					-180							
		off						-180,00001						
	<= 180,000000	on							180					
		off								180,000001				
		in	135	135	135	135					135	135	135	135
Дней	>= 1	on									1			
		off										0		
	<= 3	on											3	
		off												4
		in	2	2	2	2	2	2	2	2				
Язык			RU	US	RU	US	RU	RU	US	RU	US	RU	RU	US
Осадки			no	yes	yes	no	no	no	yes	yes	no	no	no	yes
Детализация			days	hours	days	days	hours	days	hours	days	days	hours	days	hours
Ожидаемый результат			Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска

6. При необходимости в таблицу можно добавить дополнительные отрицательные проверки. Главное придерживаться правила: не комбинировать невалидные значения. Один тест – 1 невалидное значения, остальные – валидные.

			Номер теста												
Параметр	Границы		1	2	3	4	5	6	7	8	9	10	11	12	13
Широта	>= -90,000000	on	-90												
		off		-90,000001											
	<= 90,000000	on			90										
		off				90,000001									
		in					-65	-65	-65	-65	-65	-65	-65	-65	-65
Долгота	>= -180,000000	on					-180								
		off						-180,00001							
	<= 180,000000	on							180						
		off								180,000001					
		in	135	135	135	135					135	135	135	135	135
Дней	>= 1	on									1				
		off										0			
	<= 3	on											3		
		off												4	
		in	2	2	2	2	2	2	2	2					2
Язык			RU	US	RU	US	RU	RU	US	RU	US	RU	RU	US	BY
Осадки			no	yes	yes	no	no	no	yes	yes	no	no	no	yes	yes
Детализация			days	hours	days	days	hours	days	hours	days	days	hours	days	hours	hours
Ожидаемый результат			Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Показывает прогноз	Прогноз не найден. Уточните параметры поиска	Прогноз не найден. Уточните параметры поиска

В итоге вместо 21 теста получилось 13, при этом проверяются и границы, и значения внутри и вне интервалов, а также негативные кейсы.

Важно помнить!

Негативные тесты ни в коем случае нельзя объединять друг с другом!

Представим, что мы проверяем вместе значение меньше минимального с добавлением пробела в начале». Мы предполагаем, что оба эти условия должны вызывать ошибку. По итогу, если одно из условий по факту вызывает ошибку, а второе нет, то на экране мы увидим ошибку. И ложно подумаем, что тест был пройден успешно. А во втором условии у нас будет баг, который мы не обнаружим.

Именно поэтому негативные тесты нельзя объединять друг с другом.

Также во избежание эффекта пестицида, при повторе тестов использовать разные эквивалентные значения.

7. Сценарий использования (Use Case Testing) — это перечень действий, сценарий по которому пользователь взаимодействует с приложением, программой для выполнения какого-либо действия для достижения конкретной цели. Тестирование по юзкейсам проводится для того чтобы обнаружить дополнительные логические дыры и баги в приложении, которые сложно найти в тестировании индивидуальных модулей, частей приложения отдельно друг от друга. Юзкейс тестирование может проводится как часть Приемочного тестирования.

Основное отличие между use-cases и тест-кейсом заключается в том, что use-cases описывает более широкий контекст использования системы, тогда как тест-кейс является более узким вариантом, который описывает только тестирование конкретной функции или возможности системы. Use-cases может включать множество тест-кейсов внутри себя, а также другие элементы, такие как пользовательский интерфейс, отчетность и взаимодействие с другими системами.

Пример юз-кейса, например, с онлайн-магазином. Рассмотрим сценарий «Оформление заказа»:

Название: Оформление заказа; Акторы: Покупатель, Система; Краткое описание: Покупатель оформляет заказ на товары в онлайн-магазине; Предусловия: Покупатель авторизован в системе, добавил товары в корзину.

Основной поток:

1. Покупатель переходит на страницу корзины.
2. Система отображает список товаров в корзине.
3. Покупатель выбирает адрес доставки и метод оплаты.
4. Система запрашивает данные для доставки и оплаты у Покупателя.
5. Покупатель вводит данные для доставки и оплаты.
6. Система проверяет корректность введенных данных.
7. Система формирует заказ и отправляет уведомление на email Покупателя и в службу доставки.
8. Покупатель получает уведомление об успешном оформлении заказа Ожидаемый результат: Заказ оформлен и отправлен на доставку.

В этом примере, юз-кейс помогает определить все шаги, необходимые для оформления заказа в онлайн-магазине, а также определить предусловия и ожидаемый результат.

8. ADHOC (свободное) testing — вид тестирования, который выполняется без подготовки к тестам, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование. Он не требует никакой документации, планирования, процессов которых следует придерживаться в выполнении. Также на данный вид тестирования не пишутся тест-кейсы, что в свою очередь может вызвать определенные затруднения в попытках воспроизвести дефект в системе. Такой вид зачастую может дать сходу больше результата чем тестирование по заранее определенным сценариям. Это обусловлено тем, что тестировщик на первых шагах приступает к

тестированию основного функционала и выполняет нестандартные проверки, точнее некоторые из его проверок будут нестандартными.

Часто *интуитивное* тестирование путают с *исследовательским.* Если говорить об **ad-hoc testing** и исследовательском тестировании. **Ad-hoc testing** — это более интуитивное и беспорядочное тестирование, когда тестировщик просто идет и проверяет, что ему хочется. У него нет определенной цели, структуры тестов в голове, какой-то системы. В свою очередь «**Исследовательское тестирование»** более структурированное. Обычно тестировщик знает, что ему нужно проверить, у него в голове есть цель и какая-то система проведения тестов. Хоть тесты в этом случае не обязательно должны быть оформлены в виде тест кейсов.

Чаще всего такое тестирование выполняется, когда мало времени на точное и последовательное тестирование. При этом тестировщик полагается на свое общее представление о приложении и здравый смысл

Тестирование ad-hoc имеет смысл только в случае если тестировщик владеет общей информацией о продукте. Если человек совсем не будет знать продукт, то потратит время на его изучение, особенно если проект очень сложный и большой. Поэтому нужно хорошее представление о целях проекта, его назначению и основным функциям, и возможностям. А дальше уже можно приступать к **ad-hoc testing.**

Виды интуитивного тестирования:

- **buddy testing** (совместное тестирование) — когда 2 человека, как правило разработчик + тестировщик, работают параллельно и находят дефекты в одном и том же модуле. Такой вид тестирования помогает тестировщику выполнять необходимые проверки, а программисту фиксить баги на ранних этапах.
- **pair testing** (парное тестирование) — когда 2 тестировщика проверяют один модуль и помогают друг другу. К примеру, один может искать дефекты, а второй их документировать. Таким образом, у одного тестера будет функция, скажем так обнаружителя, у другого – описателя.

Различия между buddy testing и pair testing:

1. Совместное тестирование — это сочетание юнит тестирования и системного тестирования между разработчиком и тестировщиком
2. Парное тестирование выполняется только тестировщиками с разным уровнем знаний и опыта (такое сочетание поможет поделиться взглядами и идеями)

- **monkey testing** — произвольное тестирование программы с целью ее сломать.

9. Предугадывание ошибок - в данной методике (типологически относится к тестированию **черного ящика**) нет какого-то специфического метода идентификации ошибок. Тестировщик действует, исходя из своего опыта и интуиции, пытаясь предугадать проблемные места в приложении. Поэтому успешность этой методики сильно зависит от опыта, скиллов и глубины понимания тестируемого продукта. Поэтому лучше и быстрее с такой задачей справляются QA-инженеры с опытом 1 год и более.

Методика направлена на поиск багов, не поддающихся другим методикам черного ящика. Поэтому стандартно выполняется уже после них.

Частыми ошибками в приложениях (следовательно, наиболее вероятными ошибками в тестируемом приложении) являются, например:

- Ввод некорректных (*невалидных*) параметров и символов.
- Например, ввод пробела в “цифровые” поля, где это недопустимо.
- Ошибка-исключение *null pointer exception.*
- Деление на ноль.
- Превышение максимального количества передаваемых файлов.
- И подобные «типичные пользовательские» ошибки.

Итак, цель предугадывания — найти ошибки, которые трудно “выловить” другими методиками черного ящика. Готовятся тест-кейсы, направленные на области *наиболее вероятных* ошибок; по возможности покрываются все проблемные места; но без избыточности.

Методика, разумеется, не является идеальной, способной обеспечить высокое качество — ведь она основана на интуиции, поэтому ограничена по умолчанию, зависит от умений и, главное, опытности QA-инженера. Она вовсе не гарантирует высокого *покрытия*, и должна сочетаться с другими методиками, дополнять их.

КЛИЕНТ СЕРВЕРНАЯ АРХИТЕКТУРА:

«Клиент - сервер» (англ. *client–server*) - вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных (например, передача файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных) или в виде сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, её размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики её оборудования и программного обеспечения, её также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Роль клиента и сервера:

Характеристика *клиент-сервер* описывает отношения взаимодействующих программ в приложении. Серверный компонент предоставляет функцию или услугу одному, или нескольким клиентам, которые инициируют запросы на такие услуги. Серверы классифицируются по предоставляемым ими услугам. Например, веб-сервер обслуживает веб-страницы, а файловый сервер обслуживает компьютерные файлы. Общий ресурс может быть любой из программного обеспечения и электронных компонентов компьютера — сервера, от программ и данных в процессорах и запоминающих устройств. Совместное использование ресурсов сервера представляет собой *услугу*.

Является ли компьютер клиентом, сервером или и тем, и другим, определяется характером приложения, которому требуются сервисные функции. Например, на одном компьютере могут одновременно работать веб-серверы и программное обеспечение файлового сервера, чтобы обслуживать разные данные для клиентов, отправляющих различные типы запросов. Клиентское программное обеспечение также может взаимодействовать с серверным программным обеспечением на том же компьютере. Связь между серверами, например, для синхронизации данных, иногда называется *межсерверной*.

Взаимодействие клиента и сервера:

Вообще говоря, служба — это абстракция компьютерных ресурсов, и клиенту не нужно беспокоиться о том, как сервер работает при выполнении запроса и доставке ответа. Клиенту нужно только понять ответ, основанный на известном протоколе приложения, то есть содержание и форматирование данных для запрашиваемой услуги.

Клиенты и серверы обмениваются сообщениями в шаблоне запрос-ответ. Клиент отправляет запрос, а сервер возвращает ответ. Этот обмен сообщениями является примером межпроцессного взаимодействия. Для взаимодействия компьютеры должны иметь общий язык, и они должны следовать правилам, чтобы и клиент, и сервер знали, чего ожидать. Язык и правила общения определены в протоколе связи. Все протоколы клиент-серверной модели работают на уровне приложений. Протокол прикладного уровня определяет

основные шаблоны диалога. Чтобы ещё больше формализовать обмен данными, сервер может реализовать интерфейс прикладного программирования (API). API — это уровень абстракции для доступа к сервису. Ограничивая связь определённым форматом контента, он облегчает синтаксический анализ. Абстрагируя доступ, он облегчает межплатформенный обмен данными.

Сервер может получать запросы от множества различных клиентов за короткий период времени. Компьютер может выполнять только ограниченное количество задач в любой момент и полагается на систему планирования для определения приоритетов входящих запросов от клиентов для их удовлетворения. Чтобы предотвратить злоупотребления и максимизировать доступность серверное программное обеспечение может ограничивать доступность для клиентов. Атаки типа «отказ в обслуживании» используют обязанности сервера обрабатывать запросы, такие атаки действуют путем перегрузки сервера чрезмерной частотой запросов. Шифрование следует применять, если между клиентом и сервером должна передаваться конфиденциальная информация.

Клиент – локальный компьютер (приложение, браузер) на стороне виртуального пользователя, который выполняет отправку запроса к серверу для возможности предоставления данных или выполнения определенной группы системных действий.

Сервер – очень мощный компьютер или специальное системное оборудование, которое предназначается для разрешения определенного круга задач по процессу выполнения программных кодов. Он выполняет работы сервисного обслуживания по клиентским запросам, предоставляет пользователям доступ к определенным системным ресурсам, сохраняет данные или БД.

Особенности такой модели заключаются в том, что пользователь отправляет определенный запрос на сервер, где тот системно обрабатывается и конечный результат отсылается клиенту. В возможности сервера входит одновременное обслуживание сразу нескольких клиентов.

Если одновременно поступает более одного запроса, то такие запросы устанавливаются в определенную очередь и сервером выполняются по очереди. Порой запросы могут иметь свои собственные приоритеты. Часть запросов с более высокими приоритетами будут постоянно выполняться в первоочередном порядке!

На старте развития сервиса все компоненты (Frontend, Backend, база данных) могут находиться на одном сервере. Если нагрузка растет, его можно масштабировать вертикально: поменять конфигурацию сервера на более мощную или быстро добавить ресурсов в облачный сервер — добавить число vCPU или объем памяти.

На какое-то время этого будет достаточно, но в конечном итоге мощности сервера может не хватить, и задачи разделятся на несколько серверов. Так, фронтенд уйдет на отдельный сервер, бэкенд — на второй, а база данных будет храниться еще на одной машине. Причем каждый из серверов тоже можно «проапгрейдить» вертикально.

Горизонтальное масштабирование, при котором растет количество серверов, заточенных под одну задачу, — это следующий шаг развития инфраструктуры. Оно требуется, чтобы лучше справляться с нагрузками и достичь большей гибкости в организации инфраструктуры.

При горизонтальном масштабировании появляется необходимость в новом элементе инфраструктуры — балансировщике нагрузки, распределяющем трафик.

Балансировка применима к следующему оборудованию:

- Кластер (набор серверов, набор баз данных);
- Прокси-сервер (промежуточный сервер между пользователем интернета и серверами, откуда запрашивается информация. По сути, прокси — это посредник, фильтр или шлюз, который стоит между человеком и огромными (и не всегда безопасными) данными в сети);
- Брандмауэр (он же фаервол (firewall), он же межсетевой экран — это технологический барьер, который защищает сеть от несанкционированного или

нежелательного доступа. Проще говоря, фаервол — охранник вашего компьютера, как и антивирус);

- Коммутатор (или свитч - прибор, объединяющий несколько интеллектуальных устройств в локальную сеть для обмена данными. При получении информации на один из портов, передает ее далее на другой порт, на основании таблицы коммутации или таблицы MAC-адресов);
- Система DPI (технология проверки сетевых пакетов по их содержимому с целью регулирования и фильтрации трафика, а также накопления статистических данных. В отличие от брандмауэров, Deep Packet Inspection анализирует не только заголовки пакетов, но и полезную нагрузку, начиная со второго (канального) уровня модели OSI (Модель взаимосвязи открытых систем (**Модель OSI**) **- это** концептуальная модель, которая характеризует и стандартизирует коммуникационные функции телекоммуникационной или вычислительной системы без учета ее внутренней структуры и технологии. Его цель - обеспечение совместимости различных систем связи со стандартными протоколами связи));
- DNS-сервер (специальный компьютер, который хранит или кэширует IP-адреса сайтов и выдаёт их браузеру по запросу. То есть сервер ДНС — это книга контактов больших масштабов. При вводе домена (имя сайта) в браузер DNS-серверы автоматически устанавливают связь между доменным именем и IP-устройством, и вы попадаете на нужный ресурс),
- Сетевой адаптер (плата расширения, вставляемая в разъем материнской платы (main board) компьютера. Также существуют сетевые адаптеры стандарта PCMCIA для ноутбуков (notebook), они вставляются в специальный разъем в корпусе ноутбука. Или интегрированные на материнской плате компьютера, они подключаются по какой-либо локальной шине. Появились Ethernet сетевые карты, подключаемые к USB (Universal Serial Bus) порту компьютера).

Для чего нужен балансировщик нагрузки:

Рост числа пользователей и объема трафика влечет за собой увеличение нагрузки на инфраструктуру сервиса. Балансировщик гарантирует, что сервер не будет перегружен трафиком и данные будут эффективно перемещаться между компонентами кластера.

Отказоустойчивость — главная цель. Сложность приложений растет, количество точек отказа увеличивается, они могут располагаться на разных уровнях инфраструктуры, будь то серверы или сети. Балансировщик позволяет избегать единой точки отказа — части системы, в случае выхода из строя которой вся работа будет остановлена. Если один сервер откажет, балансировщик распределит трафик между остальными элементами инфраструктуры.

Балансировщик позволяет более **оптимально использовать ресурсы** и быстрее обслуживать запросы. Например, если у вас два сервера под базы данных, балансировщик сделает так, чтобы оба были равно нагружены.

Также балансировщик обеспечит более **плавное масштабирование инфраструктуры**: при горизонтальном росте — добавлении нового сервера в кластер — он быстро и аккуратно загрузит новое «звено» инфраструктуры.

Другая важная функция балансировщика — **защита от DDoS-атак**. Ее обеспечивает задержка ответа, когда фоновые серверы не видят клиента до подтверждения по TCP. Балансировщик нагрузки Selectel проводит исходящий трафик через специальные алгоритмы, которые фильтруют TCP ACK/FIN/RST-атаки до 99,9%.

Параметры, которые могут реализоваться **на стороне сервера**:

1. Хранение, защита и доступ к данным;
2. Работа с поступающими клиентскими запросами;
3. Процесс отправки ответа клиенту.

Параметры, которые могут реализоваться **на стороне клиента**:

1. Площадка по предоставлению пользовательского графического интерфейса;
2. Формулировка запроса к серверу и его последующая отправка;

3. Получение итогов запроса и отправка дополнительной группы команд (запросы на добавление, обновление информации, удаление группы данных).

Архитектура системы клиент-сервер формулирует принципы виртуального общения между локальными компьютерами, а все правила и принципы взаимодействия находятся внутри протокола.

Сетевой протокол – это особый набор правил, на основании которого выполняется точное взаимодействие между компьютерами внутри виртуальной сети.

ВИДЫ СЕТЕВЫХ ПРОТОКОЛОВ:

TCP/IP – совокупность протоколов передачи информации. TCP/IP – это особое обозначение всей сети, которая функционирует на основе протоколов TCP, а также IP.

TCP – вид протокола, который является связующим звеном для установки качественного соединения между 2 устройствами, передачи данных и верификации их получения. (Для того, чтобы сетевая подсистема сервера различала данные, адресованные определенным сервисам, и правильно распределяла их, в протокол TCP/IP было введено понятие **номера порта**.)

IP – протокол, в функции которого входит корректность доставки сообщений по выбранному адресу. При этом информация делится на пакеты, которые могут поставляться по-разному. (IP + порт = **сокет**).

MAC – вид протокола, на основании которого происходит процесс верификации сетевых устройств. Все устройства, которые подключены к сети Интернет, содержат свой оригинальный MAC-адрес.

ICMP – протокол, который ответственен за обмен данными, но не используется для процесса передачи информации.

UDP – протокол, управляющий передачей данных, но данные не проходят верификацию при получении. Этот протокол функционирует быстрее, чем протокол TCP.

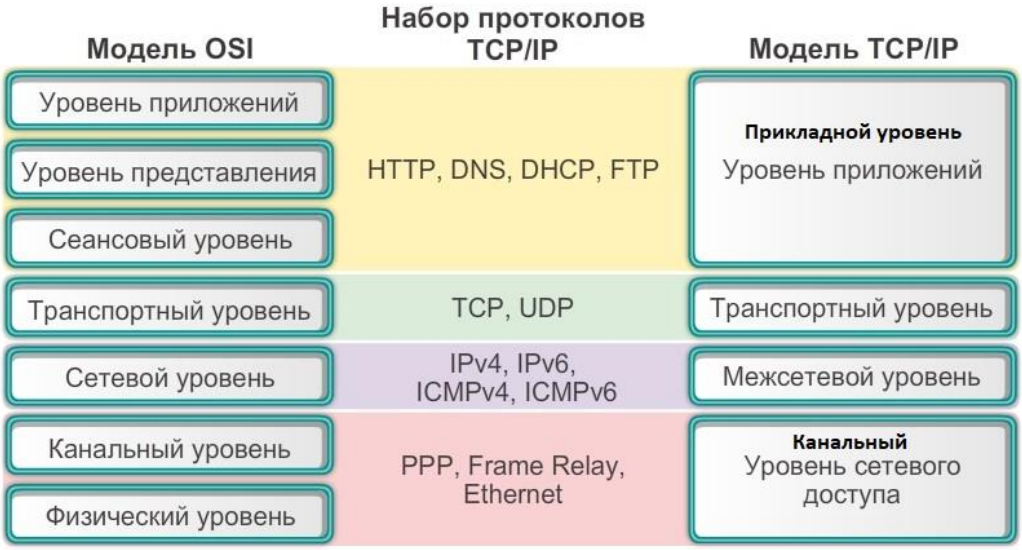
HTTP – протокол для передачи информации (гипертекста), на базе которого функционируют все сегодняшние сайты. В его возможности входит процесс запрашивания необходимых данных у виртуально удаленной системы (файлы, веб-страницы и прочее). (HTTP – 80 порт, HTTPS – 443 порт).

FTP – протокол передачи информации из особого файлового сервера на ПК конечного пользователя. (20 порт).

POP3 – классический протокол простого почтового соединения, который ответственен за передачу почты.

SMTP – вид протокола, который может устанавливать правила для передачи виртуальной почты. Он ответственен за передачу и верификацию доставки, а также оповещения о возможных ошибках. (25 порт).

Модель OSI и TCP/IP:



КОНЦЕПЦИИ ПОСТРОЙКИ КЛИЕНТ-СЕРВЕРНОЙ СИСТЕМЫ:

Слабый (тонкий) клиент – производительный сервер.

При такой модели весь процесс обработки информации перенесен на мощности сервера, а у пользователя права доступа очень строго ограничены. Сервер начинает отправлять ответ, который вообще не требует дополнительной работы по обработке. Клиент взаимодействует с пользователем: создает и отправляет запрос, принимает входящие итоги и выводит данные на экран пользователя.

Сильный (толстый) клиент.

Концепция, при которой часть обработки данных предоставляет клиенту. В такой ситуации сервер является простым хранилищем информации, а вся деятельность по обработке и предоставлению данных переносится на ПК пользователя.

ЕСТЬ СРАЗУ 2 ВИДА КЛИЕНТ-СЕРВЕРНЫХ АРХИТЕКТУР:

1. Двухуровневая, состоящая сразу из 2 узлов:

- сервер, который ответственен за получение входящих запросов и отправку ответа пользователю, применяя при этом собственные ресурсы системы;
- клиент, который может предоставлять пользовательский графический интерфейс.

Особенности работы заключаются в том, что на сервер приходит определенный запрос, потом его обрабатывают и дают напрямую, без дополнительного применения группы внешних ресурсов.

2. Трехуровневая система состоит из использования таких компонентов:

- предоставление информации – графический пользовательский, прикладной объект в виде сервера приложения;
- менеджмент ресурсов – сервер БД, который может предоставлять данные.

Особенность работы состоит в том, что сразу несколько серверов могут обрабатывать клиентские запросы. Процесс распределения операций может существенным образом снизить нагрузку на используемый сервер.

Трехуровневая архитектура может трансформироваться до многоуровневой, возможностью установки группы дополнительных серверов. Подобная виртуальная архитектура позволяет существенным образом повысить эффективность функционирования информационных систем, а также выполнить оптимизированное распределение части ее программно-аппаратных ресурсов.

Хост (от англ. **host** — «хозяин, принимающий гостей») — любое устройство, предоставляющее сервисы формата «клиент-сервер» в режиме сервера по каким-либо интерфейсам и уникально определенное на этих интерфейсах.

HTTP/HTTPS:

HTTP — широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов (то есть документов, которые могут содержать ссылки, позволяющие организовать переход к другим документам).

Аббревиатура HTTP расшифровывается как *HyperText Transfer Protocol*, «протокол передачи гипертекста». В соответствии со спецификацией OSI, HTTP является протоколом прикладного (верхнего, 7-го) уровня.

Характеристики HTTP:

1. Простой (для работы с ним используется обычный текст);

2. Расширяемый (запросы и ответы имеют Headers (заголовки являются основной частью этих HTTP-запросов и ответов, и они несут информацию о браузере клиента, запрошенной странице, сервере и многом другом), которые могут расширяться до бесконечности, для общения клиента с сервером);

3. Сессия вместо состояния (у запросов нет состояния, есть сессия, которая обычно хранится в **Cookies (Куки)** (это небольшие текстовые файлы, в которые браузер записывает

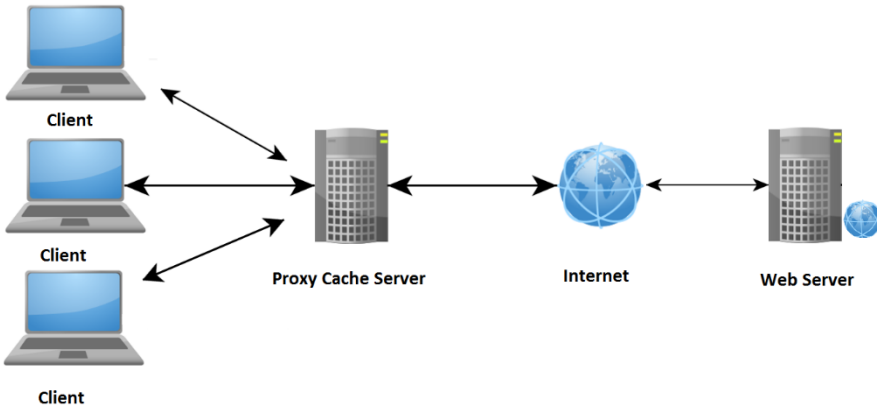
данные с посещенных вами сайтов. Файлы cookie позволяют сайтам «запоминать» своих посетителей, например, чтобы каждый раз не переспрашивать их логин и пароль).

Протокол HTTP предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает данный запрос, формирует ответ и передаёт его обратно клиенту. После этого клиентское приложение может продолжить отправлять другие запросы, которые будут обработаны аналогичным образом.

Задача, которая традиционно решается с помощью протокола HTTP — обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером. На данный момент именно благодаря протоколу HTTP обеспечивается работа Всемирной паутины.

Также HTTP часто используется как протокол передачи информации для других протоколов прикладного уровня, таких как SOAP, XML-RPC и WebDAV. В таком случае говорят, что протокол HTTP используется как «транспорт».

API многих программных продуктов также подразумевает использование HTTP для передачи данных — сами данные при этом могут иметь любой формат, например, XML или JSON.



Client* = *user agent (Браузер, Postman, SoupUI, Jmeter, Java app).

Proxy (Сетевые адаптеры, Fiddler, Charles, DataPower, SOWA). (Функции: Кеширование, фильтрация, выравнивание нагрузки (балансировщик), аутентификация (контроль доступа к ресурсам), протоколирование (логи)).

Server* = *Web server* = *Host (NGINX, Apache Tomcat).

Кэш — это один из уровней памяти устройства или программы. Это высокоскоростное буферное хранилище, в котором располагаются нужные данные. Обычно кэш небольшого размера, и в нем хранится временная информация или та, к которой обращаются чаще всего. Процесс помещения информации в кэш называется **кэшированием**. Кэш есть в компьютерах и мобильных телефонах, отдельные кэши есть у программ, например, у браузеров и веб-приложений. Он важен, потому что позволяет быстрее получать доступ к часто используемым данным, оптимизирует и ускоряет работу.

Аппаратный кэш устройства — сервера, компьютера или телефона — это специальный участок памяти с особой архитектурой. Кэш приложений и сервисов — чаще всего программный: он хранится в обычной памяти, в папках на устройстве или на отдельных серверах. Скорость доступа оптимизируется с помощью кода.

Виды веб-кэшей:

Кэш браузера (Browser cache):

Если вы изучите окно настроек любого современного веб-браузера (например, Internet Explorer, Safari или Mozilla), вы, вероятно, заметите параметр настройки «Кэш». Эта опция позволяет выделить область жесткого диска на вашем компьютере для хранения

просмотренного ранее контента. Кэш браузера работает согласно довольно простым правилам. Он просто проверяет являются ли данные “свежими”, обычно один раз за сессию (то есть, один раз в текущем сеансе браузера).

Этот кэш особенно полезен, когда пользователь нажимает кнопку “Назад” или кликает на ссылку, чтобы увидеть страницу, которую только что просматривал. Также, если вы используете одни и те же изображения навигации на вашем сайте, они будут выбираться из браузерного кэша почти мгновенно.

Прокси-кэш (Proxy cache):

Прокси-кэш работает по аналогичному принципу, но в гораздо большем масштабе. Прокси обслуживают сотни или тысячи пользователей; большие корпорации и интернет-провайдеры часто настраивают их на своих файрволах или используют как отдельные устройства (intermediaries).

Поскольку прокси не являются частью клиента или исходного сервера, но при этом обращены в сеть, запросы должны быть к ним как-то переадресованы. Одним из способов является использование настроек браузера для того, чтобы вручную указать ему к какому прокси обращаться; другой способ — использование перехвата (interception proxy). В этом случае прокси обрабатывают веб-запросы, перенаправленные к ним сетью, так, что клиенту нет нужды настраивать их или даже знать об их существовании.

Прокси-кэши являются своего рода общей кэш-памятью (shared cache): вместо обслуживания одного человека, они работают с большим числом пользователей и поэтому очень хороши в сокращении времени ожидания и сетевого трафика. В основном, из-за того, что популярный контент запрашивается много раз.

Данные между клиентом и сервером в рамках работы протокола передаются с помощью HTTP-сообщений. Они бывают двух видов:

- **Запросы (HTTP Requests)** — сообщения, которые отправляются клиентом на сервер, чтобы вызвать выполнение некоторых действий. Зачастую для получения доступа к определенному ресурсу. Основой запроса является HTTP-заголовок.
- **Ответы (HTTP Responses)** — сообщения, которые сервер отправляет в ответ на клиентский запрос.

Само по себе сообщение представляет собой информацию в текстовом виде, записанную в несколько строчек.

В целом, как **запросы** HTTP, так и **ответы имеют следующую структуру:**

1. ***Стартовая строка (start line)*** — используется для описания версии используемого протокола и другой информации — вроде запрашиваемого ресурса или кода ответа. Как можно понять из названия, ее содержимое занимает ровно одну строчку.
2. ***HTTP-заголовки (HTTP Headers)*** — несколько строчек текста в определенном формате, которые либо уточняют запрос, либо описывают содержимое *тела* сообщения.
3. ***Пустая строка***, которая сообщает, что все метаданные для конкретного запроса или ответа были отправлены.
4. Опциональное ***Тело сообщения***, которое содержит данные, связанные с запросом, либо документ (например, HTML-страницу), передаваемый в ответе.

Стартовая строка HTTP-запроса состоит из трех элементов:

1. ***Метод HTTP-запроса*** (method, реже используется термин verb). Обычно это короткое слово на английском, которое указывает, что конкретно нужно сделать с запрашиваемым ресурсом. Например, метод GET сообщает серверу, что пользователь хочет получить некоторые данные, а POST — что некоторые данные должны быть помещены на сервер.
2. ***Цель запроса.*** Представлена указателем ресурса **URL**, который ***состоит из*** протокола, доменного имени (или IP-адреса), пути к конкретному ресурсу на сервере.

Дополнительно может содержать указание порта, несколько параметров HTTP-запроса и еще ряд опциональных элементов.

3. ***Версия*** используемого протокола (либо HTTP/1.1, либо HTTP/2), которая определяет структуру следующих за стартовой строкой данных.

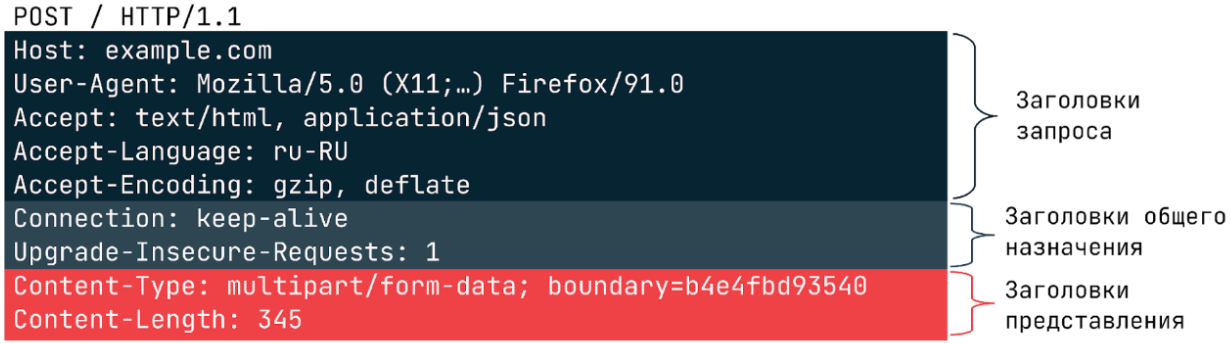
В примере ниже стартовая строка указывает, что в качестве метода используется GET, обращение будет произведено к ресурсу /index.html, по версии протокола HTTP/1.1:

Основные структурные элементы URL:

Методы:

Методы позволяют указать конкретное действие, которое мы хотим, чтобы сервер выполнил, получив наш запрос. Так, некоторые методы позволяют браузеру (который в большинстве случаев является источником запросов от клиента) отправлять дополнительную информацию в теле запроса — например, заполненную форму или документ.

Ниже приведены наиболее используемые методы и их описание:



Разберемся с каждым из названных элементов подробнее:

Метод	Описание
GET*	Позволяет запросить (получить) некоторый конкретный ресурс. Дополнительные данные могут быть переданы через строку запроса (Query String (Params)) в составе URL (например: ?param=value). О составляющих URL мы поговорим чуть позже. (Запрос не имеет тела.)

POST*	Позволяет отправить данные на сервер. Поддерживает отправку различных типов файлов, среди которых текст, PDF-документы и другие типы данных в двоичном виде. Обычно метод POST используется при отправке информации (например, заполненной формы логина) и загрузке данных на веб-сайт, таких как изображения и документы. (Запрос имеет тело.)
HEAD	Здесь придется забежать немного вперед и сказать, что обычно сервер в ответ на запрос возвращает заголовок и тело, в котором содержится запрашиваемый ресурс. Данный метод при использовании его в запросе позволит получить только заголовки, которые сервер бы вернул при получении GET-запроса к тому же ресурсу. Запрос с использованием данного метода обычно производится для того, чтобы узнать размер запрашиваемого ресурса перед его загрузкой.
OPTIONS	Позволяет запросить информацию о сервере, в том числе информацию о допускаемых к использованию на сервере HTTP-методов.
PUT*	Используется для создания, обновления (размещения) новых ресурсов на сервере. Если на сервере данный метод разрешен без надлежащего контроля, то это может привести к серьезным проблемам безопасности. (Запрос имеет тело.) (Изменение ресурса (объекта) полностью).
PATCH*	Позволяет внести частичные изменения в указанный ресурс (объект) по указанному расположению.
DELETE*	Позволяет удалить существующие ресурсы на сервере. Если использование данного метода настроено некорректно, то это может привести к атаке типа «Отказ в обслуживании» (Denial of Service, DoS) из-за удаления критически важных файлов сервера. (Запрос не имеет тела.)

GET, HEAD, PUT, DELETE, – являются **идемпотентными методами** (метод, при котором, повторный идентичный запрос имеет один и тот же результат, не меняющий состояние сервера).

URL:

Получение доступа к ресурсам по HTTP-протоколу осуществляется с помощью указателя URL. Он представляет собой строку, которая позволяет указать запрашиваемый ресурс и еще ряд параметров.

Существуют так же аббревиатуры:

- **URI -** Uniform Resource Identifier (унифицированный **идентификатор** ресурса);
- **URL -** Uniform Resource Locator (унифицированный **определитель местонахождения** ресурса);
- **URN -** Uniform Resource Name (унифицированное **имя** ресурса).

Многие считают, что http://google.com или http://yandex.ru - это просто URL-адреса, но, однако мы можем говорить о них как о URI. Фактически, URI представляет собой

расширенный набор URL-адресов и нечто, называемое URN. Таким образом, мы можем с уверенностью заключить, что все URL являются URI. Однако обратное неверно.

Твое имя, скажем, “Джон Доу” - это URN. Место, в котором вы живете, например, “Улица Вязов, 13” – это уже URL. Вы можете быть идентифицированы как уникальное лицо с вашим именем или вашим адресом. Эта уникальная личность – это уже URI. И хотя ваше имя может быть вашим уникальным идентификатором (URI), оно не может быть URL-адресом, поскольку ваше имя не помогает найти ваше местоположение. Другими словами, URI (которые являются URN) не являются URL-адресами.

Вернемся в интернет:

- **URI** – имя и адрес ресурса в сети, включает в себя URL и URN;
- **URL** – адрес ресурса в сети, определяет местонахождение и способ обращения к нему;
- **URN** – имя ресурса в сети, определяет только название ресурса, но не говорит как к нему подключиться.

Рассмотрим примеры:

- **URI** – https://wiki.merionet.ru/images/vse-chto-vam-nuzhno-znat-pro-devops/1.png
- **URL** - https://wiki.merionet.ru
- **URN** - images/vse-chto-vam-nuzhno-znat-pro-devops/1.png

Что такое URI?

URI является последовательностью символов, которая идентифицирует какой-то ресурс. URI может содержать URL и URN.

URI содержит в себе следующие части:

- **Схема (scheme)** - показывает на то, как обращаться к ресурсу, чаще всего это сетевой протокол (http, ftp, ldap).
- **Иерархическая часть (hier-part)** - данные, необходимые для идентификации ресурса (например, адрес сайта).
- **Запрос (query)** - необязательные дополнительные данные ресурса (например, поисковой запрос).
- **Фрагмент (fragment)** – необязательный компонент для идентификации вторичного ресурса (например, место на странице).

Общий синтаксис URI выглядит так:

URI = scheme ":" hier-part ["?" query] ["#" fragment]

Что такое URL?

Всегда помните - URI может содержать URL, но URL указывает только адрес ресурса.

URL содержит следующую информацию:

- Протокол, который используется для доступа к ресурсу – http, https, ftp.
- Расположение сервера с использованием IP-адреса или имени домена - например, wiki.merionet.ru - это имя домена. https://192.168.1.17 - здесь ресурс расположен по указанному IP-адресу.
- Номер порта на сервере. Например, http://localhost: 8080, где 8080 - это порт.
- Точное местоположение в структуре каталогов сервера. Например - https://wiki.merionet.ru/ip-telephoniya/ - это точное местоположение, если пользователь хочет перейти в раздел про телефонию на сайте.
- Необязательный идентификатор фрагмента. Например, https://www.google.com/search?ei=qw3eqwe12e1w&q=URL, где q = URL - это строка запроса, введенная пользователем.

Общий синтаксис URL выглядит так:

URL = [protocol]://www.[domain_name]:[port 80]/[path or exaction resource location]?[query]#[fragment]

Использование URL неразрывно связано с другими элементами протокола, поэтому далее мы рассмотрим его основные компоненты и строение:

Поле **Scheme** используется для указания используемого протокола, всегда сопровождается (заканчивается) двоеточием и двумя косыми чертами (://).

Host указывает местоположение ресурса, в нем может быть, как доменное имя, так и IP-адрес.

Port, как можно догадаться, позволяет указать номер порта, по которому следует обратиться к серверу. Оно начинается с двоеточия (:), за которым следует номер порта. При отсутствии данного элемента номер порта будет выбран по умолчанию в соответствии с указанным значением **Scheme** (например, для http:// это будет порт 80).

Далее следует поле **Path (путь, ссылка) (Path Params)**. Оно указывает на ресурс, к которому производится обращение. Если данное поле не указано, то сервер в большинстве случаев вернет указатель по умолчанию (например: index.html).

Виды путей (ссылок):

1. Абсолютные ссылки (absolute):

href="http://sites.ru/shop/" — ссылка на главную страницу магазина
href="http://sites.ru/shop/t-shirts/t-shirt-life-is-good/" — ссылка на страницу товара

2. Относительные ссылки (relative):

При использовании относительных ссылок за точку отсчета каждый раз берется отправная страница.

href="t-shirts/t-shirt-life-is-good/" — ссылка с главной страницы на страницу товара
href="../../" — ссылка со страницы товара на главную страницу

Здесь можно сделать первый вывод. Хотя относительные адреса выглядят короче абсолютных, однако абсолютные адреса предпочтительнее, так как одну и ту же ссылку можно применять в неизменном виде на любой странице сайта, на какой бы глубине она не находилась.

Поле **Query String (Query Params)** начинается со знака вопроса (?), за которым следует пара «параметр-значение», между которыми расположен символ равно (=). В поле Query String могут быть переданы несколько параметров с помощью символа амперсанд (&) в качестве разделителя.

Не все компоненты необходимы для доступа к ресурсу. Обязательно следует указать только поля **Scheme** и **Host**.

Версии HTTP:

Последняя стабильная, наиболее стандартизированная версия протокола первого поколения (версия HTTP/1.1) вышла в далеком 1997 году.

У HTTP/1.1 есть ряд значительных недостатков:

- Заголовки, в отличие от тела сообщения, передавались в несжатом виде.
- Часто большая часть заголовков в сообщениях совпадала, но они продолжали передаваться по сети.
- Отсутствовала возможность так называемого мультиплексирования — механизма, позволяющего объединить несколько соединений в один поток данных. Приходилось открывать несколько соединений на сервере для обработки входящих запросов.

С выходом HTTP/2 было предложено следующее решение: HTTP/1.X-сообщения разбивались на так называемые *фреймы*, которые встраивались в поток данных.

Фреймы данных (тела сообщения) отделялись от фреймов заголовка, что позволило применять сжатие. Вместе с появлением потоков появился и ранее описанный механизм мультиплексирования — теперь можно было обойтись одним соединением для нескольких потоков.

Единственное, о чем стоит сказать в завершение темы: HTTP/2 перестал быть текстовым протоколом, а стал работать с «сырой» двоичной формой данных. Это ограничивает чтение и создание HTTP-сообщений «вручную». Однако такова цена за возможность реализации более совершенной оптимизации и повышения производительности.

Заголовки (Headers):

HTTP-заголовок представляет собой строку формата «Имя-Заголовок:Значение», с двоеточием(:) в качестве разделителя. Название заголовка не учитывает регистр, то есть между Host и host, с точки зрения HTTP, нет никакой разницы. Однако в названиях заголовков принято начинать каждое новое слово с заглавной буквы.

В запросах может передаваться большое число различных заголовков, но все их можно разделить на три категории:

1. **Общие (General)**, которые применяются ко всему сообщению целиком. (URL, метод, статус код, Remote address (самый последний удаленный адрес с которого был произведен запрос на сервер), Refer Policy).
2. **Заголовки запроса** уточняют некоторую информацию о запросе, сообщая дополнительный контекст или ограничивая его некоторыми логическими условиями.
3. **Заголовки сущности (представления) (Entity, Custom)**, которые описывают формат данных сообщения и используемую кодировку. Добавляются к запросу только в тех случаях, когда с ним передается некоторое тело. (x-client-data)

Ниже можно видеть пример заголовков в запросе:

Заголовок	Описание
Host	Используется для указания того, с какого конкретно хоста запрашивается ресурс. В качестве возможных значений могут использоваться как доменные имена, так и IP-адреса. На одном HTTP-сервере может быть размещено несколько различных веб-сайтов. Для обращения к какому-то конкретному требуется данный заголовок.
Content-Length	Размер сообщения, если запрос имеет тело (высчитывается автоматически).
User-Agent	Заголовок используется для описания клиента, который запрашивает ресурс. Он содержит достаточно много информации о пользовательском окружении. Например, может указать, какой браузер используется в качестве клиента, его версию, а также операционную систему, на которой этот клиент работает.
Refer	Используется для указания того, откуда поступил текущий запрос. Например, если вы решите перейти по какой-нибудь ссылке в этой статье, то вероятнее всего к запросу будет добавлен заголовок Refer: https://selectel.ru

Accept	Позволяет указать, какой тип медиафайлов принимает клиент. В данном заголовке могут быть указаны несколько типов, перечисленные через запятую (‘ , ‘). А для указания того, что клиент принимает любые типы, используется следующая последовательность — */*.
Cookie	Данный заголовок может содержать в себе одну или несколько пар «Куки-Значение» в формате cookie=value. **Куки** представляют собой небольшие фрагменты данных, которые хранятся как на стороне клиента, так и на сервере, и выступают в качестве идентификатора. Куки передаются вместе с запросом для поддержания доступа клиента к ресурсу. Помимо этого, куки могут использоваться и для других целей, таких как хранение пользовательских предпочтений на сайте и отслеживание клиентской сессии. Несколько кук в одном заголовке могут быть перечислены с помощью символа точка с запятой (‘ ; ‘), который используется как разделитель.
Authorization	Используется в качестве еще одного метода идентификации клиента на сервере. После успешной идентификации сервер возвращает токен, уникальный для каждого конкретного клиента. В отличие от куки, данный токен хранится исключительно на стороне клиента и отправляется клиентом только по запросу сервера. Существует несколько типов аутентификации, конкретный метод определяется тем веб-сервером или веб-приложением, к которому клиент обращается за ресурсом.

Тело запроса:

Завершающая часть HTTP-запроса — это его тело. Не у каждого HTTP-метода предполагается наличие тела. Так, например, методам вроде GET, HEAD, DELETE, OPTIONS обычно не требуется тело. Некоторые виды запросов могут отправлять данные на сервер в теле запроса: самый распространенный из таких методов — POST.

Ответы HTTP:

HTTP-ответ является сообщением, которое сервер отправляет клиенту в ответ на его запрос. Его структура равна структуре HTTP-запроса: стартовая строка, заголовки, пустая строка и тело.

Строка статуса (Status line):

Стартовая строка HTTP-ответа называется **строкой статуса** (status line). На ней располагаются следующие элементы:

1. Уже известная нам по стартовой строке запроса ***версия протокола*** (HTTP/2 или HTTP/1.1).
2. ***Код состояния***, который указывает, насколько успешно завершилась обработка запроса.
3. ***Пояснение*** — короткое текстовое описание к коду состояния. Используется исключительно для того, чтобы упростить понимание и восприятие человека при просмотре ответа.

Так выглядит строка состояния ответа.

Коды состояния и текст статуса:

Коды состояния HTTP используются для того, чтобы сообщить клиенту статус их запроса. HTTP-сервер может вернуть код, принадлежащий одной из пяти категорий кодов состояния:

Категория	Описание
1xx	(Информационные) Коды из данной категории носят исключительно информативный характер и никак не влияют на обработку запроса.
2xx	(Успешные) Коды состояния из этой категории возвращаются в случае успешной обработки клиентского запроса.
3xx	(Перенаправление) Эта категория содержит коды, которые возвращаются, если серверу нужно перенаправить клиента.
4xx	(Ошибка клиента) Коды данной категории означают, что на стороне клиента был отправлен некорректный запрос. Например, клиент в запросе указал не поддерживаемый метод или обратился к ресурсу, к которому у него нет доступа.
5xx	(Ошибка сервера) Ответ с кодами из этой категории приходит, если на стороне сервера возникла ошибка.

Полный список кодов состояния доступен в спецификации к протоколу, ниже приведены только самые распространенные коды ответов:

Категория	Описание
200 OK	Возвращается в случае успешной обработки запроса, при этом тело ответа обычно содержит запрошенный ресурс. (GET)

201 Created	Код ответа об успешном статусе указывает, что запрос выполнен успешно и привел к созданию ресурса. (POST)
204 No content	Запрос обработан успешно, но возвращать данные не требуется. Также новая или обновленная информация может быть возвращена в ответе, но в итоге она не будет отличаться относительно того, что было первоначально отправлено на сервер и, таким образом, считается, что клиент уже имеет актуальную информацию.
302 Found	Перенаправляет клиента на другой URL. Например, данный код может прийти, если клиент успешно прошел процедуру аутентификации и теперь может перейти на страницу своей учетной записи.
400 Bad Request	Данный код можно увидеть, если запрос был сформирован с ошибками. Например, в нем отсутствовали символы завершения строки.
401 Unauthorized	При открытии страницы сайта означает неверную авторизацию или аутентификацию пользователя на стороне сервера при обращении к определенному url-адресу. Чаще всего она возникает при ошибочном вводе имени и/или пароля посетителем ресурса при входе в свой аккаунт.
403 Forbidden	Означает, что клиент не обладает достаточными правами доступа к запрошенному ресурсу. Также данный код можно встретить, если сервер обнаружил вредоносные данные, отправленные клиентом в запросе.
404 Not Found	Каждый из нас, так или иначе, сталкивался с этим кодом ошибки. Данный код можно увидеть, если запросить у сервера ресурс, которого не существует на сервере.
405 Method Not Allowed	Указанный клиентом метод нельзя применить к текущему ресурсу.
500 Internal Error	Данный код возвращается сервером, когда он не может по определенным причинам обработать запрос.
504 Gateway Timeout	Появляется, когда в течение заданного периода времени один сервер не получает своевременный ответ от другого сервера, который действует как шлюз или прокси

Заголовки ответа:

Response Headers, или заголовки ответа, используются для того, чтобы уточнить ответ, и никак не влияют на содержимое тела. Они существуют в том же формате, что и остальные заголовки, а именно «Имя-Значение» с двоеточием (:) в качестве разделителя.

Ниже приведены наиболее часто встречаемые в ответах заголовки:

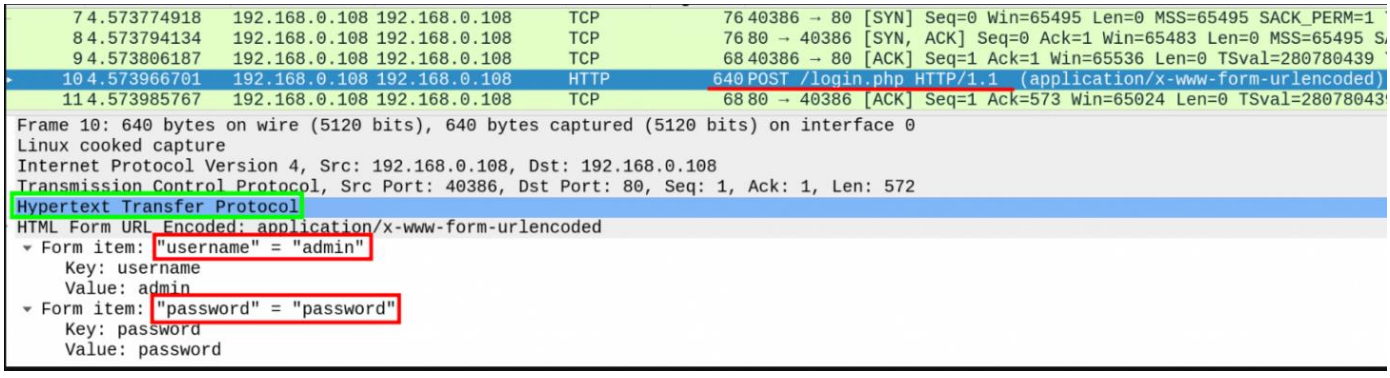
Категория	Пример	Описание
Server	Server: ngnix	Содержит информацию о сервере, который обработал запрос.
Set-Cookie	Set-Cookie:PHPSSID=bf42938f	Содержит куки, требуемые для идентификации клиента. Браузер парсит куки и сохраняет их в своем хранилище для дальнейших запросов.
WWW-Authenticate	WWW-Authenticate: BASIC realm=»localhost»	Уведомляет клиента о типе аутентификации, который необходим для доступа к запрашиваемому ресурсу.
Content-Type	text/html	Тип возвращаемого содержимого.

Тело ответа:

Последней частью ответа является его тело. Несмотря на то, что у большинства ответов тело присутствует, оно не является обязательным. Например, у кодов «**201 Created (успешно создано)» или «204 No Content (отсутствует объект)» тело отсутствует**, так как достаточную информацию для ответа на запрос они передают в заголовке.

Безопасность HTTP-запросов, или что такое HTTPs:

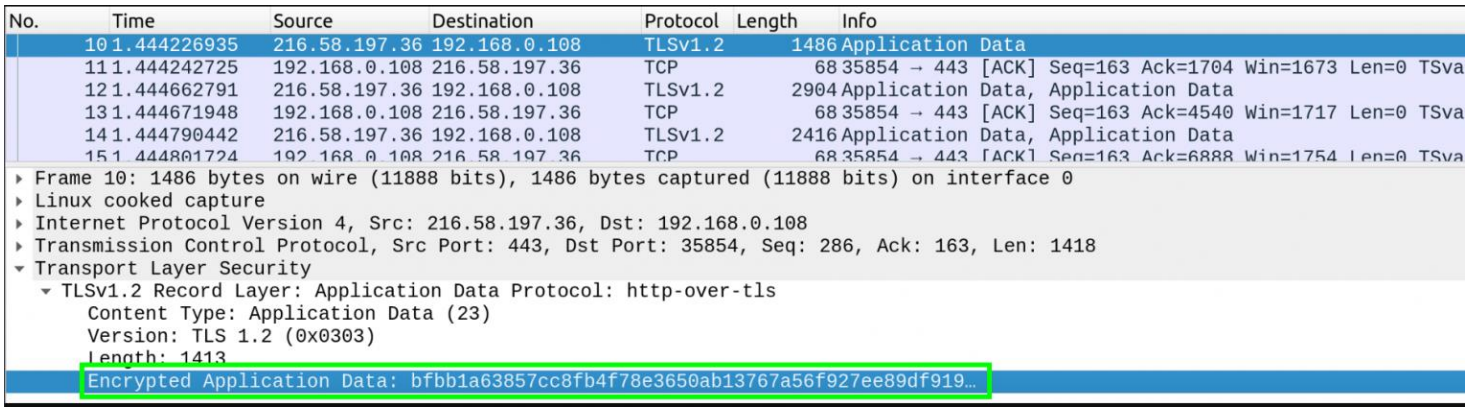
HTTP является расширяемым протоколом, который предоставляет огромное количество возможностей, а также поддерживает передачу всевозможных типов файлов. Однако, вне зависимости от версии, у него есть один существенный недостаток, который можно заметить если перехватить отправленный HTTP-запрос:



Да, все верно: данные передаются в открытом виде. HTTP сам по себе не предоставляет никаких средств шифрования.

HTTPs (HyperText Transfer Protocol, secure) является расширением HTTP-протокола, который позволяет шифровать отправляемые данные, перед тем как они попадут на транспортный уровень. Данный протокол по умолчанию ***использует порт 443***.

Теперь если мы перехватим не HTTP, а HTTPs-запрос, то не увидим здесь ничего интересного:



Данные передаются в едином зашифрованном потоке, что делает невозможным получение учетных данных пользователей и прочей критической информации средствами обычного перехвата.

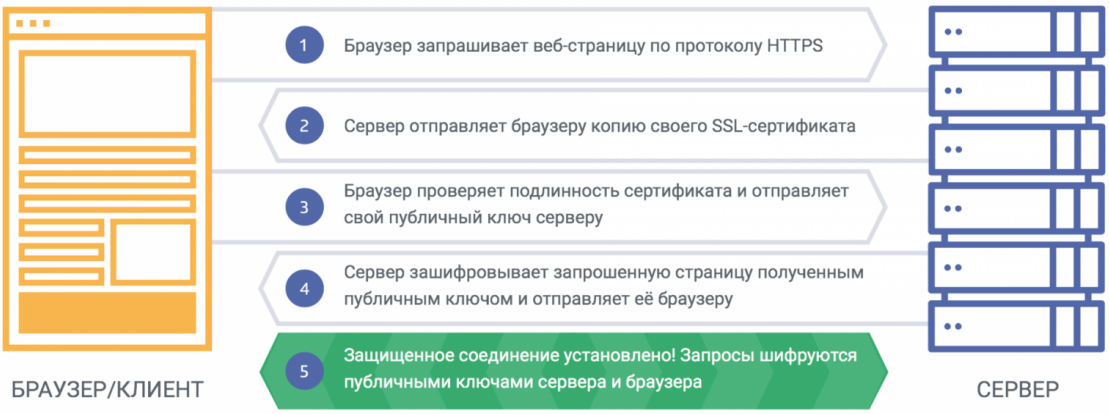
SSL (***secure sockets layer*** — уровень защищённых сокетов) представляет собой криптографический протокол для безопасной связи. С версии 3.0 SSL заменили на **TLS** (***transport layer security*** — безопасность транспортного уровня), но название предыдущей версии прижилось, поэтому сегодня под SSL чаще всего подразумевают TLS.

Цель протокола — обеспечить защищенную передачу данных. При этом для аутентификации используются асимметричные алгоритмы шифрования (пара открытый — закрытый ключ), а для сохранения конфиденциальности — симметричные (секретный ключ). Первый тип шифрования более ресурсоемкий, поэтому его комбинация с симметричным алгоритмом помогает сохранить высокую скорость обработки данных.

Рукопожатие:

Когда пользователь заходит на веб-сайт, браузер запрашивает информацию о сертификате у сервера, который высылает копию SSL-сертификата с открытым ключом. Далее, браузер проверяет сертификат, название которого должно совпадать с именем веб-сайта.

Кроме того, проверяется дата действия сертификата и наличие корневого сертификата, выданного надежным центром сертификации. Если браузер доверяет сертификату, то он генерирует предварительный секрет (pre-master secret) сессии на основе открытого ключа, используя максимально высокий уровень шифрования, который поддерживают обе стороны.



Сервер расшифровывает предварительный секрет с помощью своего закрытого ключа, соглашается продолжить коммуникацию и создать общий секрет (master secret), используя определенный вид шифрования. Теперь обе стороны используют симметричный ключ, который действителен только для данной сессии. После ее завершения ключ уничтожается, а при следующем посещении сайта процесс рукопожатия запускается сначала.

Ключи бывают:

- Публичный/ Открытый ключ – может зашифровывать.
- Приватный/Закрытый ключ – может расшифровывать.

API:

API (англ. Application Programming Interface — программный интерфейс приложения) — это набор способов и правил, по которым различные программы общаются между собой и обмениваются данными.

Все эти взаимодействия происходят с помощью функций, классов, методов, структур, а иногда констант одной программы, к которой обращаются другие. Это основной принцип работы API.

Программный интерфейс похож на договор между клиентом и продавцом. Только клиентом выступает приложение, которому нужны данные, а продавцом — сервер или ресурс, с которого мы эти данные берём. В таком договоре прописываются условия того, как и какие данные может получить клиент.

Интеграционное тестирование = тестирование API.

Есть у любого API:

- Точка входа (адрес (обычно это URL (например, название сайта) и порт), на который посылаются сообщения называется **Endpoint**. В русскоговорящем сегменте интернета иногда используется термин «Ручка» - **сделать запрос к Endpoint - дёрнуть за ручку**);
- Данные на вход;
- Данные на выход.

API встречается практически везде:

- В языках программирования он помогает функциям корректно общаться друг с другом. Вызывающая функция должна соблюдать тип данных и последовательность параметров вызываемой функции.
- В операционной системе он помогает программам получать данные из памяти или менять настройки ОС. Поэтому, чтобы разрабатывать приложения под конкретную операционную систему, нужно знать её API.
- В вебе сервисы общаются друг с другом через программный интерфейс. Если API открытый, то официальную документацию по работе с ним публикуют создатели сервиса-источника.

Интерфейс — это граница между двумя функциональными системами, на которой происходит их взаимодействие и обмен информацией. При этом процессы внутри каждой из систем скрыты друг от друга.

С помощью интерфейса можно использовать возможности разных систем, не задумываясь о том, как они обрабатывают наши запросы и что у них «под капотом». Например, чтобы позвонить, не обязательно знать, как смартфон обрабатывает нажатия на тачскрин. Важно лишь, что в гаджете есть «кнопка», которая всегда возвращает одинаковый результат в ответ на определённые действия.

Точно так же с помощью вызовов API можно выполнить определённые функции программы, не зная, как она работает. Поэтому API и называют интерфейсом.

Возможности API:

- **Предоставляет доступ к готовым инструментам.** Например, к функциям библиотеки для машинного обучения TensorFlow — они помогают быстро создать нейросеть, не тратя время на разработку инструментов с нуля.
- **Повышает безопасность.** API позволяет вынести в отдельное приложение функциональность, которая должна быть защищена. Так снижается вероятность некорректного использования этих функций другими программами.
- **Связывает разные системы.** Если вам нужно подключить к сайту платёжную систему или авторизацию через соцсети, без API не обойтись.
- **Снижает стоимость разработки.** Часто бывает, что дешевле воспользоваться платным API, чем создавать функциональность с нуля.

Никаких специальных правил или ограничений на набор функций для API нет. Разработчики включают в него те методы, которые, по их мнению, будут полезны для взаимодействия клиентских приложений с их сервисом.

REST API — это архитектурный подход (стиль), который устанавливает ограничения для API: как они должны быть устроены и какие функции поддерживать. Это позволяет стандартизировать работу программных интерфейсов, сделать их более удобными и производительными.

Слово ***REST*** — акроним от «REpresentational State Transfer», что переводится на русский как ***«передача состояния представления», «передача репрезентативного состояния» или «передача самоописываемого состояния».***

В отличие от, например, SOAP API, REST API — не протокол, а простой список рекомендаций, которым можно следовать или не следовать. Поэтому у него нет собственных методов. С другой стороны, его автор Рой Филдинг создал ещё и протокол HTTP, так что они очень хорошо сочетаются, и REST обычно используют в связке с HTTP. Хотя новичкам нужно помнить: **REST — это не только HTTP, а HTTP — не только REST**.

Иногда, помимо акронима REST, можно встретить слово **RESTful**. Это не термин, но в сообществе его применяют к веб-сервисам, которые соответствуют REST-архитектуре. RESTful используют как прилагательное.

Как работает REST API: 6 принципов архитектуры (RESTful):

Всего в REST есть шесть требований к проектированию API. Пять из них обязательные, одно — опциональное:

- Клиент-серверная модель (client-server model).
- Отсутствие состояния (statelessness).
- Кэширование (cacheability).
- Единообразие интерфейса (uniform interface).
- Многоуровневая система (layered system).
- Код по требованию (code on demand) — необязательно.

REST основан на доступе к ресурсам. **Ресурсы** — это любые данные: текст, изображение, видео, аудио, целая программа. Например, HTML веб-страницы, тоже ресурс. На сервере четко обозначен ресурс, который отвечает за обработку запроса.

У ресурса должен быть идентификатор: URI = URL + URN (описано ранее).

Клиент и сервер общаются неким представлением о ресурсе (Тип данных – любой (JSON, XML, текст, HTML, YAML).

REST управляются гиперссылками в представлении. (По ответу понятно, как можно дальше взаимодействовать с API, куда перейти для получения большей информации).

1. Клиент-серверная модель:

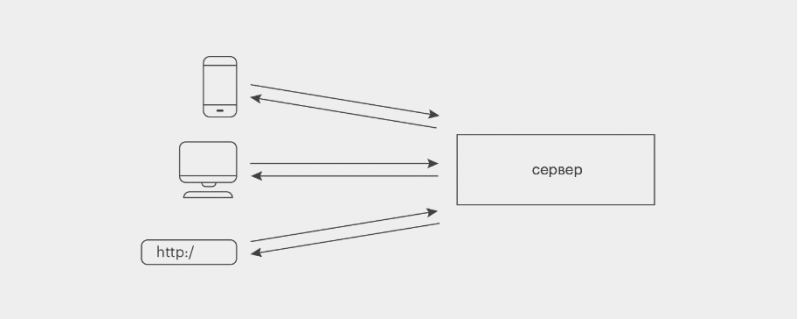
Клиент и сервер не зависимы друг от друга.

Представим, что вы делаете сервис для учёта деловых переписок. Сами переписки хранятся на сервере, а доступ к ним можно получить из мобильного приложения. Оно

не будет хранить никаких данных — только отправлять запросы на сервер, получать ответы и отображать их на экране смартфона.

Если вы когда-нибудь захотите полностью изменить логику работы сервера, то это никак не отразится на мобильном приложении. До тех пор, пока они понимают запросы и ответы друг друга, конечно.

А чтобы дать доступ к сервису из десктопного приложения и личного сайта, достаточно написать два новых клиента — а на сервере ничего менять не надо. Такая вот гибкость.



2. Отсутствие состояния:

Второй принцип настолько важен, что даже отражён в названии архитектурного стиля — **Representational State Transfer (*«передача самоописываемого состояния»*)**. Это значит, что на сервере не хранится никаких данных о прошлых взаимодействиях с клиентом — каждый запрос должен содержать всю информацию для его обработки.

Например, кто-то запросил последнее сообщение от ООО «Рога и копыта». В этом запросе содержится вся информация, которая нужна серверу, чтобы дать корректный ответ.

Если клиент потом хочет получить предпоследнее сообщение, то он не может просто сказать: «Дай мне соседний ресурс» — ему нужно заново составить полный запрос по всем правилам.

Это снижает нагрузку на сервер, что особенно полезно, если к нему подключено одновременно много клиентов. Не нужно хранить дополнительную информацию о прошлых обращениях каждого из них. Достаточно обработать каждый запрос в отдельности.

3. Кэширование:

Иногда клиент запрашивает с сервера одни и те же данные по несколько раз — например, вы постоянно обращаетесь к какому-нибудь важному письму в сервисе для учёта деловых переписок.

Если при каждом таком запросе сервер будет с нуля собирать нужные данные и отправлять их клиенту, нагрузка на систему повысится — особенно когда таких повторов много. Решением проблемы в REST API стало **кэширование**, то есть сохранение части данных у клиента или на промежуточных серверах.

Однако тут тоже важно подойти к делу без излишнего фанатизма и не кэшировать всю информацию подряд. Во-первых, для этого потребовались бы слишком большие объёмы памяти. Во-вторых, какие-то данные (скажем, количество исходящих писем) со временем могут устаревать — зачем же держать этот неактуальный хлам в кэше? Именно поэтому в каждом ответе сервера на запрос есть пометка о том, можно ли его кэшировать.

4. Единообразие интерфейса:

Должен быть **единый способ обращения** к каждому ресурсу. Например, мы хотим добавить в наш сервис новую функциональность для просмотра данных о денежных переводах. Понятно, что логика интерфейса для обращения к ним должна быть такой же, как и для всего, что было в сервисе раньше.

Файлы обычно передаются клиенту не в том виде, в котором хранятся на сервере. В вебе их часто преобразуют в **JSON*** (но может быть любой другой тип данных) и только потом отправляют клиенту. Ответ на запросы к новому ресурсу должен приходить в том же формате, что и к старым, и сразу же содержать дополнительную информацию: что разрешается делать с ресурсом, можно ли его изменять и удалять на сервере и так далее.

Для реализации единообразного интерфейса в REST API используется принцип **HATEOAS** (Hypermedia as the Engine of Application State).

***JSON** (JavaScript Object Notation, нотация объектов JavaScript) - это текстовый формат обмена данными. Он представлен наборами пар "ключ-значение", причем ключ - это всегда строка, а значение может задаваться одним из следующих типов:

- число;
- строка;
- логическое значение;
- массив;
- объект;
- нулевое значение null.

Несколько важных правил JSON:

- В формате данных JSON ключи прописываются в двойных кавычках.
- Ключ и значение разделяются двоеточием (:).
- Может быть несколько пар "ключ-значение". Каждая пара отделяется запятой (,).
- В данных JSON недопустимы комментарии (// или /* */). (Но при желании это ограничение можно обойти).

Можно сгенерировать JSON Schema – используя онлайн приложения.

5. Многоуровневая система:

До сих пор мы рассматривали сервер как единую сущность. Но его структура куда сложнее. Между ним и клиентом есть несколько промежуточных узлов, выполняющих вспомогательные функции, — **прокси-серверы**.

Они используются для кэширования, обеспечения безопасности, дополнительной обработки данных. Если основных серверов несколько, то дополнительные серверы-балансировщики могут распределять нагрузку между ними и решать, в какой из них направлять запрос:

Никто из участников цепочки не знает всего пути, который проходит запрос, — только своих «соседей» справа и слева. Ни клиент, ни один из прокси-серверов не знает, к кому он обращается — к основному сервису или к другому прокси. В REST API это работает в обе стороны: никакие серверы (ни основные, ни прокси) не знают, кому отправляют ответ и уходит ли он куда-то дальше.

6. Код по требованию (необязательно):

Этот принцип означает, что сервер в ответ на запрос может **отправить исходный код**, который выполняется уже на стороне клиента. Благодаря этому можно передавать целые сценарии. Например, динамические элементы пользовательского интерфейса, написанные на JavaScript.

В REST API требование необязательно, потому что не всем сайтам и сервисам нужно умение работать с готовыми скриптами.

Для чего используют REST API:

Архитектурный стиль REST — самый распространённый подход к проектированию API. Вот в каких случаях его применяют:

- пропускная способность соединения с сервером ограничена;
- нужно соединить мобильные приложения с серверными;

- проект разбит на микросервисы;
- сервис предоставляет свои возможности другим разработчикам;
- используется AJAX (аббревиатура, которая означает Asynchronous Javascript and XML);
- известно, что систему нужно будет масштабировать.

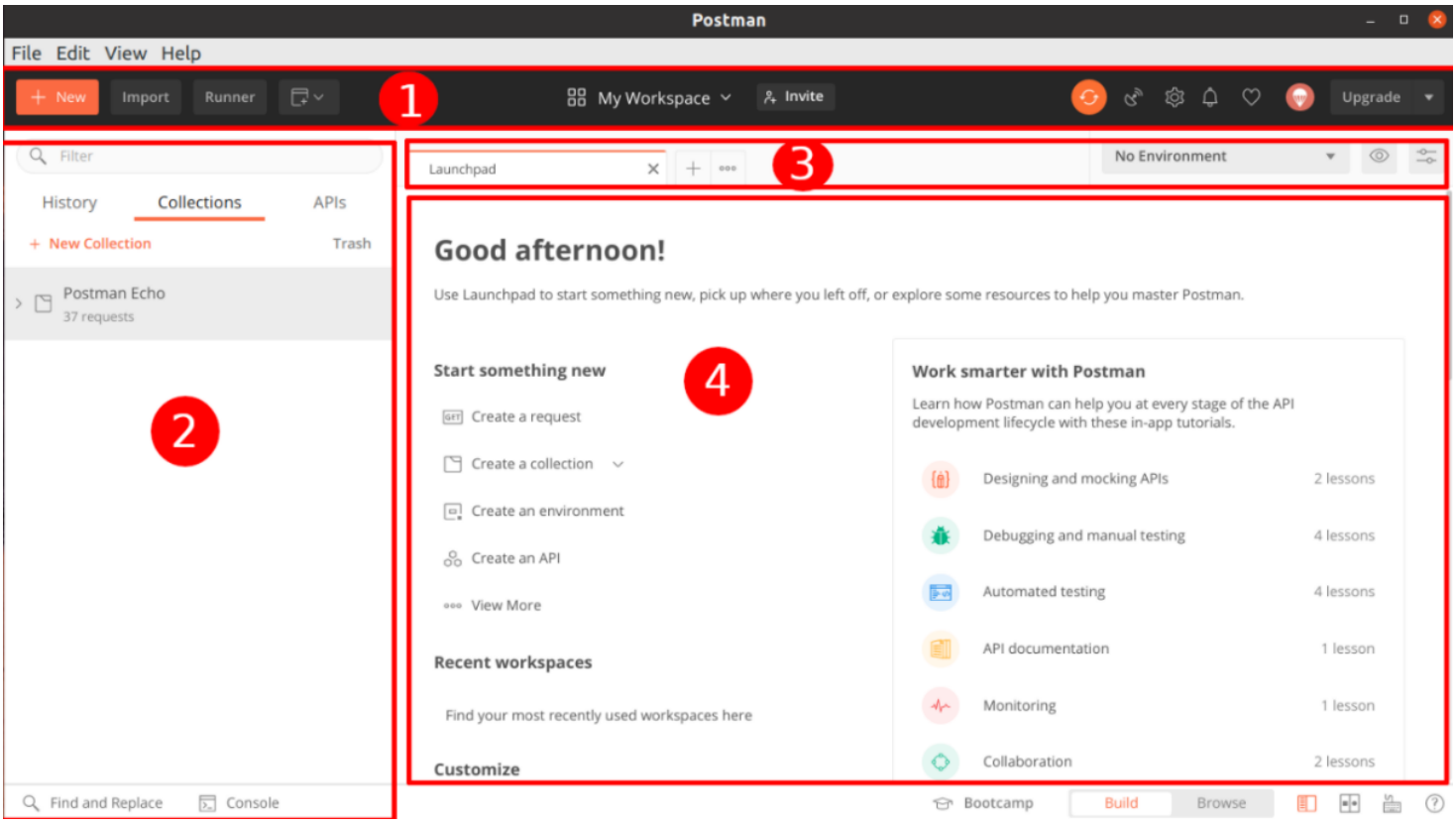
Документация REST API – описывается в отдельной документации или в SWAGGER файле (Swagger Tools: Swagger или OpenAPI framework состоит из 4 основных компонентов: **Swagger Core** - позволяет генерировать документацию на основе существующего кода основываясь на Java Annotation. **Swagger Codegen** - позволит генерировать клиентов для существующей документации. **Swagger UI** - красивый интерфейс, который представляет документацию. Дает возможность просмотреть какие типы запросов есть, описание моделей и их типов данных. **Swagger Editor** - Позволяет писать документацию в YAML или JSON формата.).

Для работы с REST чаще используют POSTMAN.

Тестирование интерфейса API проводится путем анализа точности выходных данных в зависимости от подаваемых при входном запросе. Этим и занимается Postman: он составляет и отправляет их на указанные URL, получает обратно и сохраняет в базе данных. При желании возможно сохранение типовых запросов в коллекции (для быстрого доступа) и создание для них разного окружения.

Интерфейс приложения Postman:

Главное окно программы разделено на четыре области. Разделение на блоки идет по функционалу, что заметно упрощает настройку и управление.

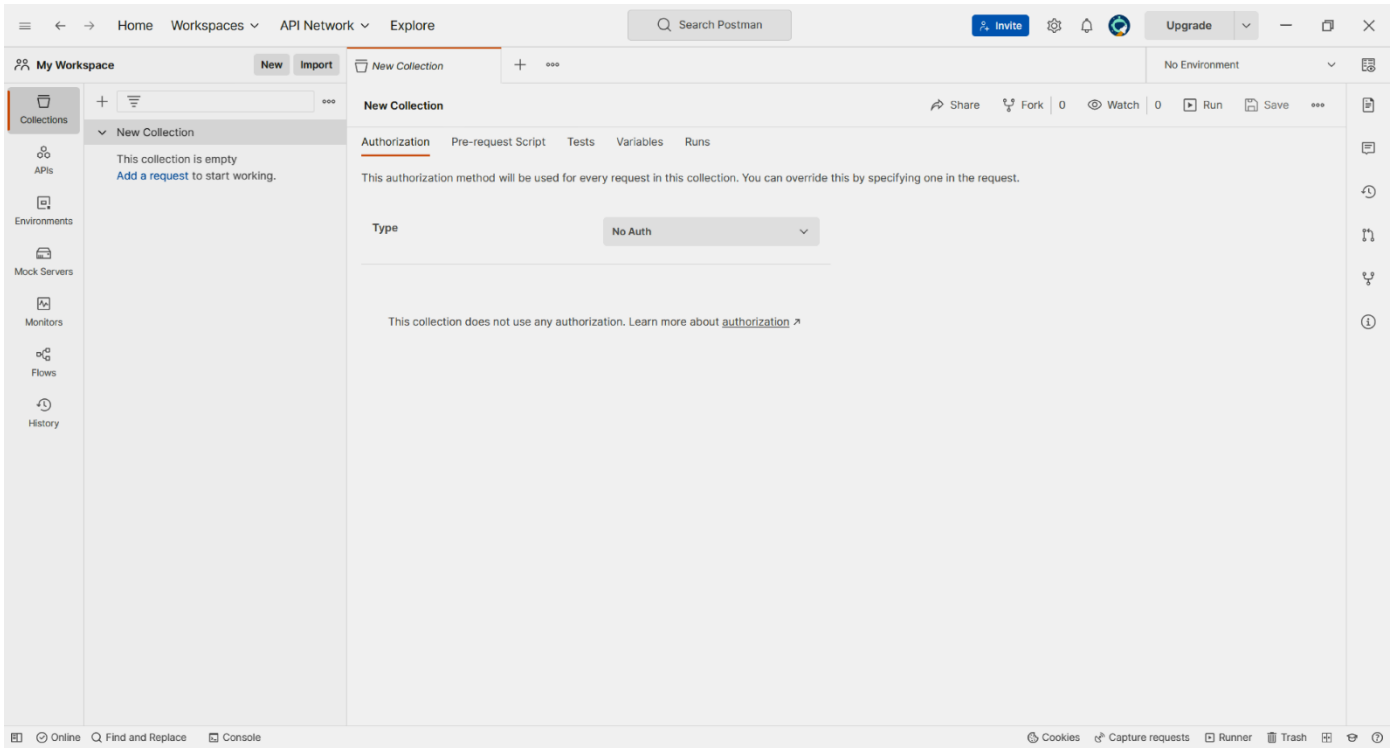


Описание меню:

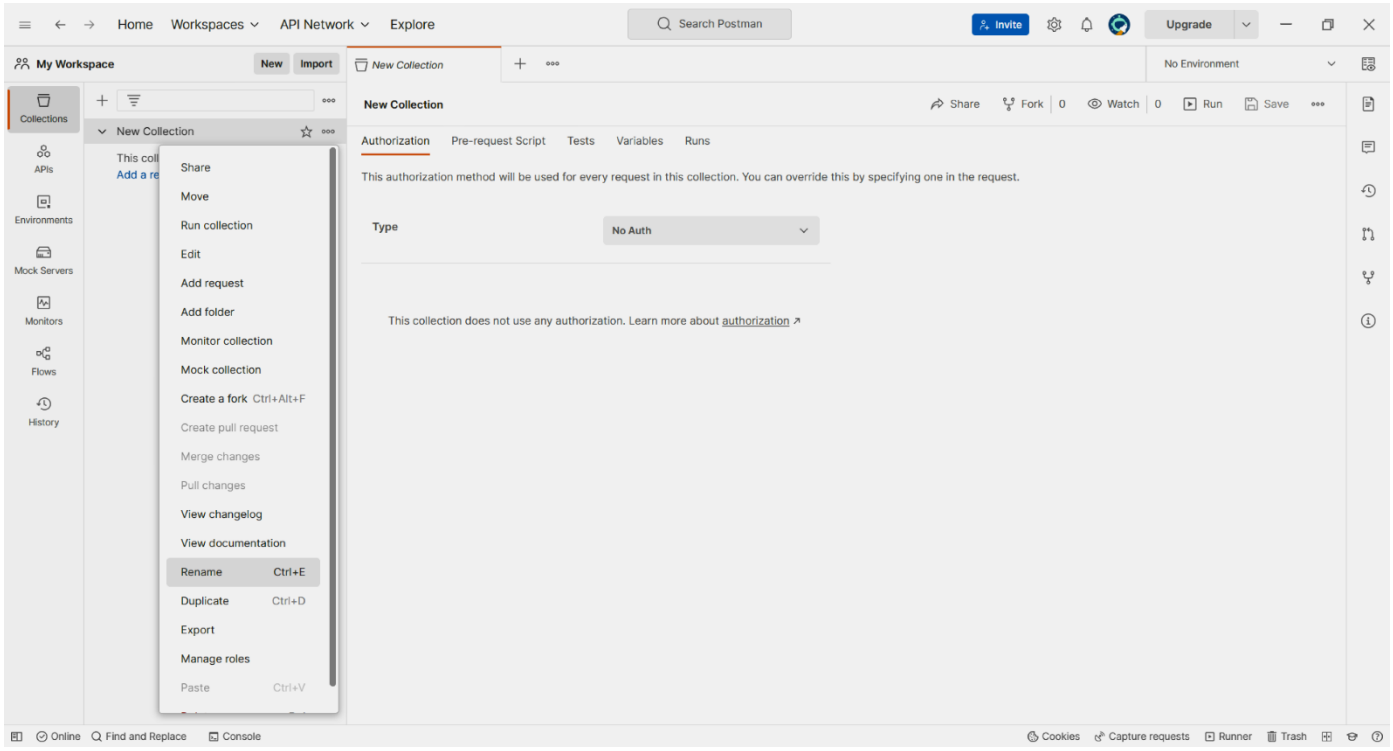
1. Верхняя панель – здесь расположены основные настройки программы.
2. Боковая панель – сюда выделены запросы, выполненные ранее или сохраненные в качестве «избранного».
3. Панель вкладок – инструмент переключения между разными запросами.
4. Рабочая область – все базовые настройки отправленного запроса, перечень возвращаемых по нему данных.

Как создать коллекцию:

Чтобы создать коллекцию, нужно перейти во вкладку **Collections** и нажать плюс. В левой части приложения появится пункт **New Collection**.



Изменим название на **Test Collection**. Для этого нажмём на три точки справа от **New Collection** и во всплывающем окошке выберем пункт **Rename**.



В рабочей области коллекции есть пять вкладок:

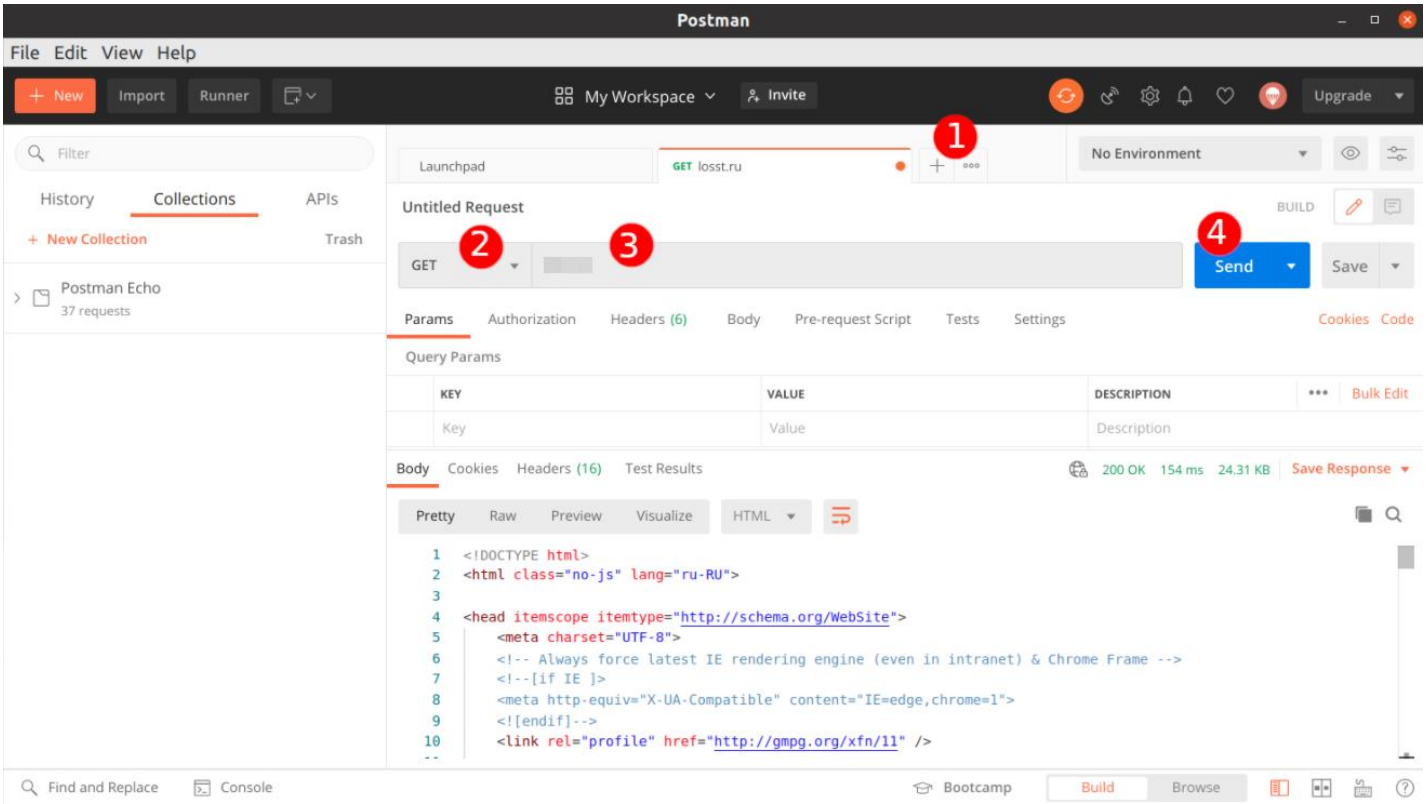
- **Authorization.** Здесь можно настроить метод и параметры авторизации, которые будут использоваться в каждом запросе внутри коллекции.
- **Pre-request Script**. Здесь можно написать программу на JavaScript, которая будет выполняться перед каждой отправкой запроса внутри коллекции. Для наиболее

распространённых алгоритмов Postman предлагает готовые сниппеты. Их можно использовать, чтобы не писать код с нуля.

- **Tests.** Работает как Pre-request Script, но выполняет код после выполнения запроса. Именно этот раздел используют тестировщики для проверки API. Здесь тоже есть готовые сниппеты.
- **Variables.** Здесь можно создать переменную и присвоить ей значение. Потом эту переменную можно использовать, указав её название в двойных фигурных скобках {{имя переменной}}.
- **Runs.** Postman позволяет запускать запросы не по отдельности, а все сразу — внутри одной коллекции или папки. В разделе Runs хранится информация о таких прогонах и результатах их тестов.

Выполнение запроса:

Выполнение простого запроса, без сохранения в коллекции, осуществляется кликом по кнопке со значком <+>. В результате откроется новая вкладка, где есть возможность выбрать тип запроса (GET или POST) и внести домен, который будет открываться. Остается нажать на кнопку **Send**, которая и запустит процедуру проверки.



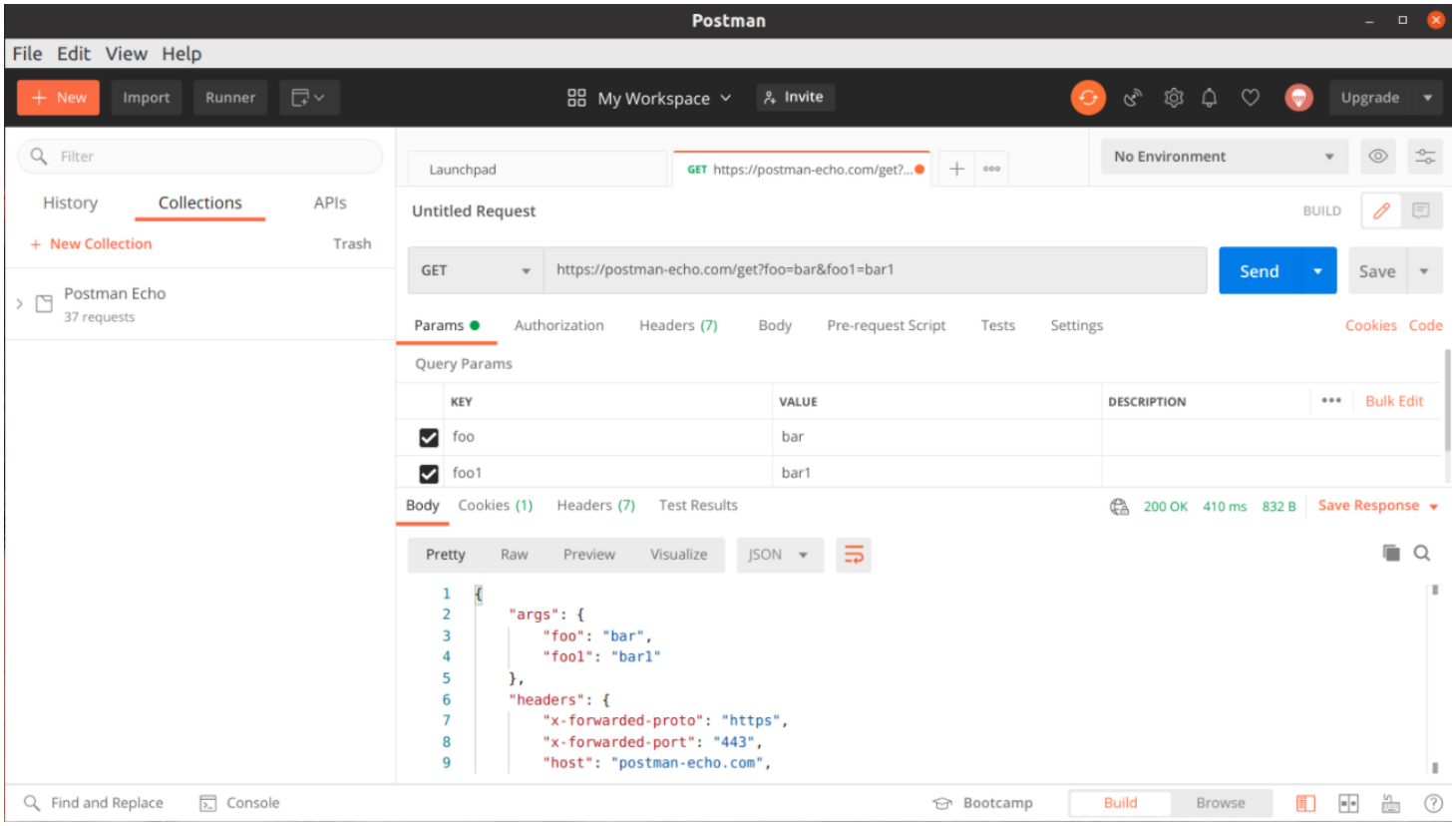
В нижней части страницы появится код страницы (HTML). Здесь имеется несколько вкладок:

1. Body – данные, содержащиеся в теле запроса.
2. Cookie – информация, записанная сервером.
3. Headers – заголовки, которые были возвращены.

На первой вкладке, где отображается тело запроса, есть выбор нескольких вариантов отображения. Так, **Pretty** интересна для получения JSON-данных – программа отформатирует их в достаточно удобном формате. Если выбрать режим **Raw**, информация будет представлена «как есть», без каких-либо изменений. Вкладка **Preview** отображает сайт в том виде, в котором он открывается в браузере.

Передача параметров в Postman:

В программу встроен собственный сервис API, который и используется для тестирования внешних ресурсов. Чтобы обратиться к нему, следует кликнуть на «плюсик», выбрать из выпадающего списка тип запроса GET, а вместо домена вставить ссылку на сервис https://postman-echo.com/get.

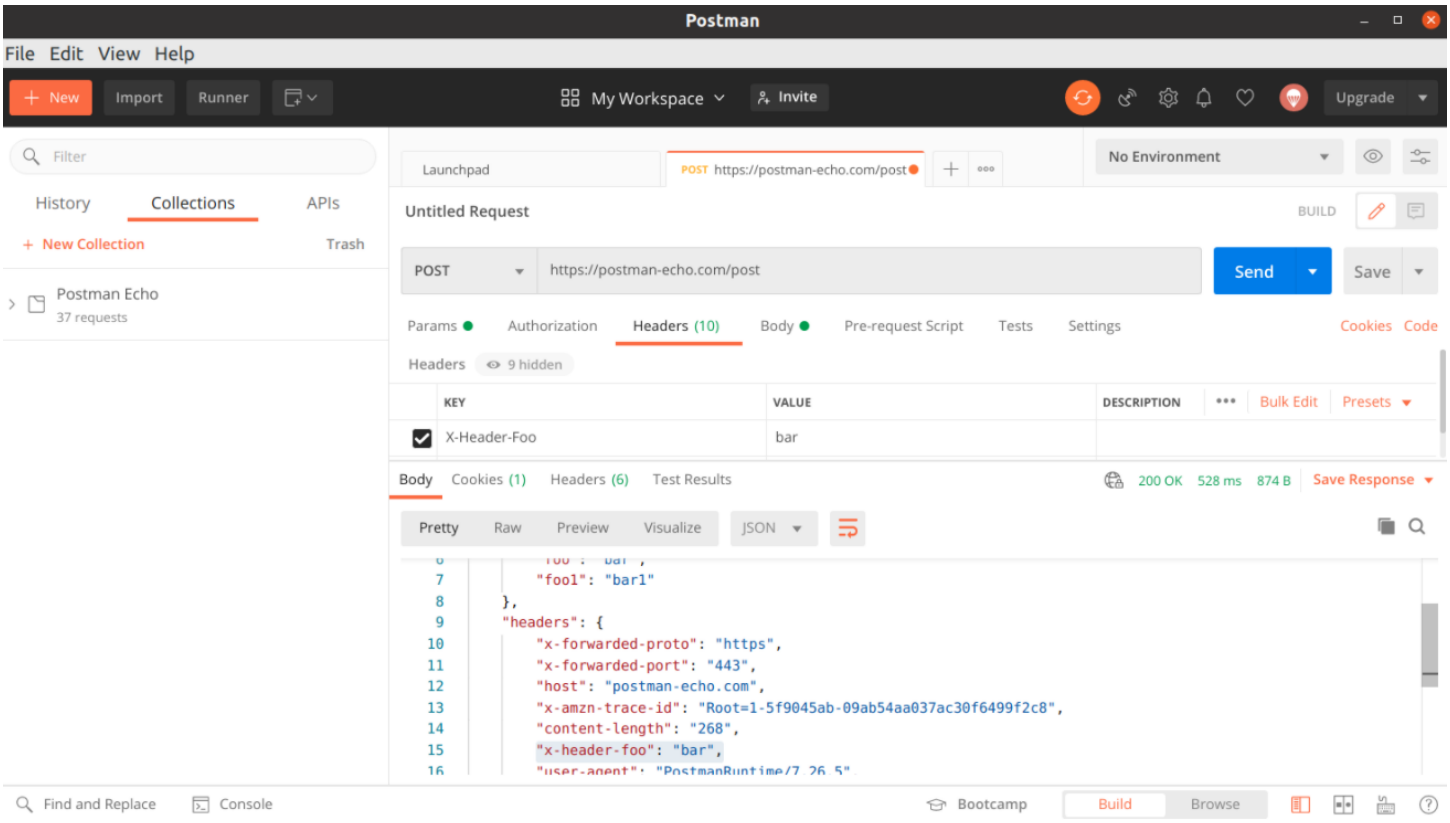


Затем нужно открыть вкладку **Params** и в разделе **Query Params** под строкой **Key** внести название отправляемого параметра. Следом под строкой **Value** нужно написать еще одно значение. Количество не ограничено – пользователь вносит столько параметров, сколько ему нужно для тестирования конкретного API.

Остается нажать на кнопку **Send** и получить ответ на отправленные запросы. Чтобы потом параметры не задействовались при тестировании реально существующих веб-ресурсов, достаточно снять с них галочки. Это укажет программе, что нужно отправлять запросы без учета внесенных параметров.

Передача параметров формы и заголовков:

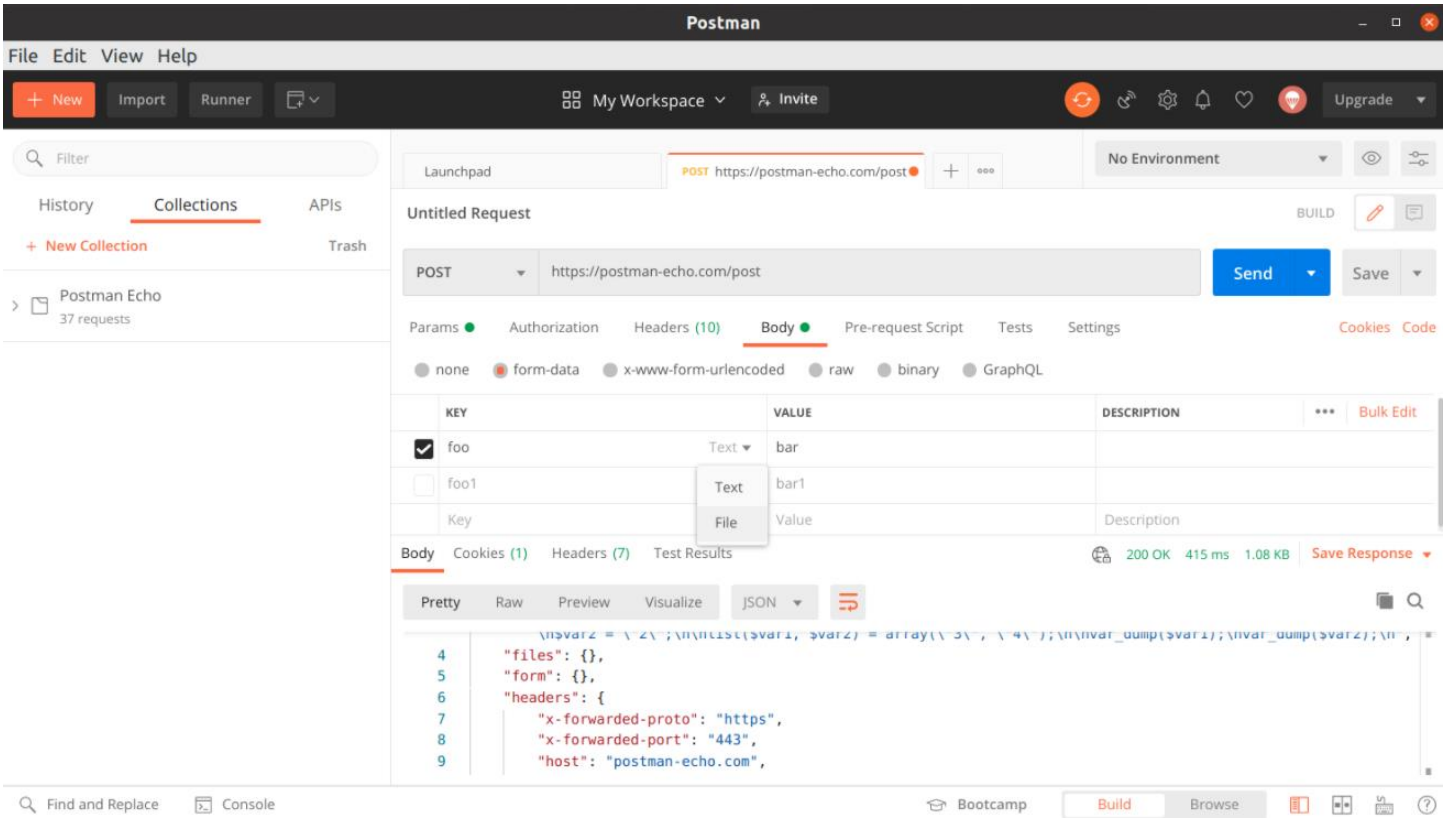
В отличие от GET, запрос POST передается не в ссылке на сайт, а в теле запроса. Чтобы проверить работоспособность программы, используется обращение к адресу https://postman-echo.com/post. Во время настройки на вкладке **Body** нужно включить режим **form-data**, затем внести схожие параметры и нажать на кнопку **Send**.



Если взаимодействие по API требует передачи токенов авторизации, понадобится привлечь к этому HTTP-заголовки. Такой формат работы используется, например, в движке Xenforo, написанном на PHP для развертывания форумов. Для передачи в заголовке какой-либо информации нужно зайти на вкладку **Headers** и добавить любое имя со значением (на выбор пользователя). После отправки информации внизу окна будет отображен ответ сервера.

Передача файла в Postman:

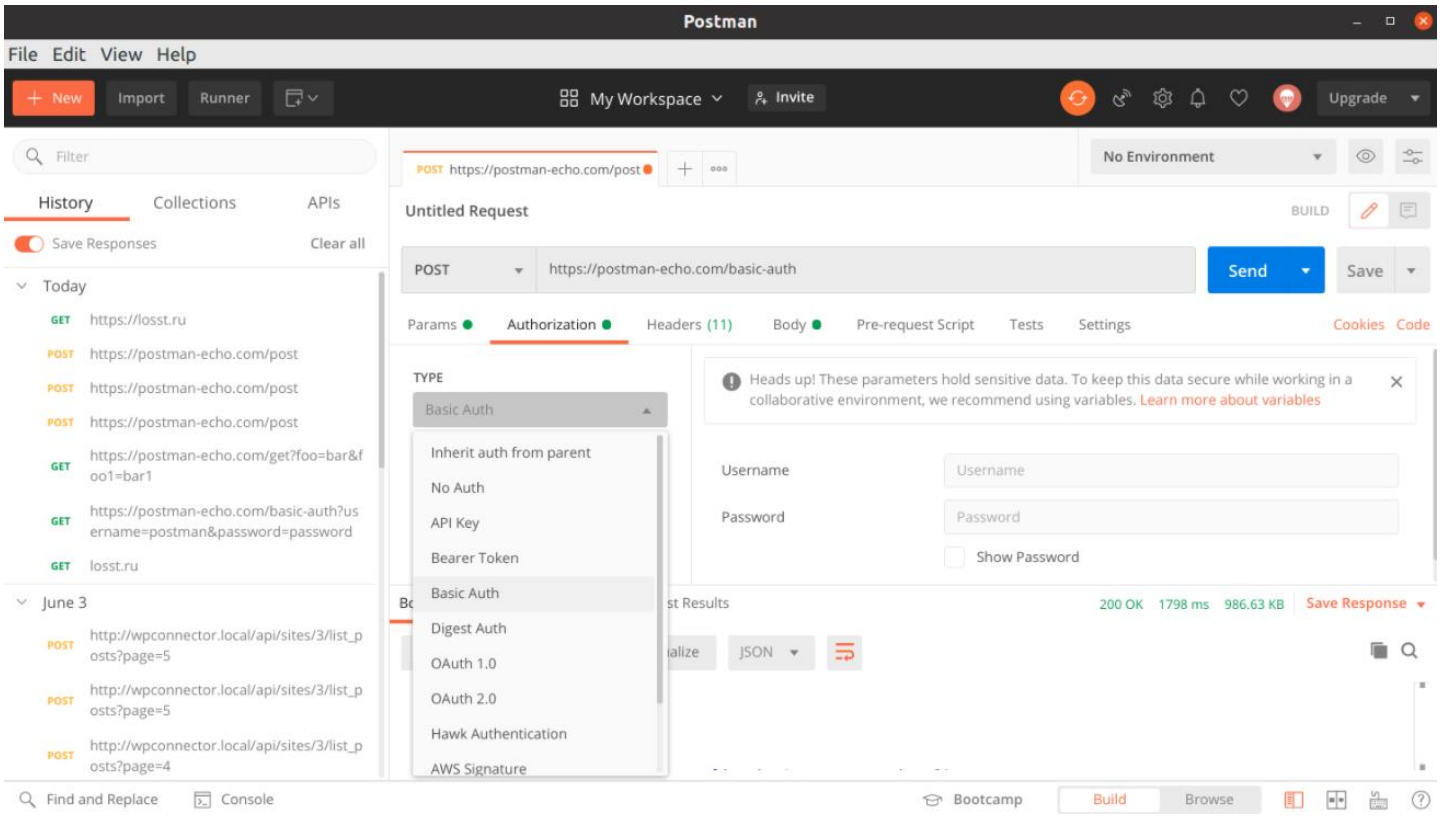
Программа Postman позволяет отправлять файлы, а не только текстовые данные, как в приведенных выше примерах. Чтобы сделать это, достаточно перейти на вкладку **Body**, зайти в раздел **form-data** и выбрать тип параметра **File** (вместо **Text**).



Затем следует нажать на кнопку **Select File** и выбрать отправляемый файл. После отправки данных на сервер он будет виден в секции **files**. Ничего сложного в процедуре нет, приведенная выше схема работает со всеми типами файлов.

Авторизация Basic Auth:

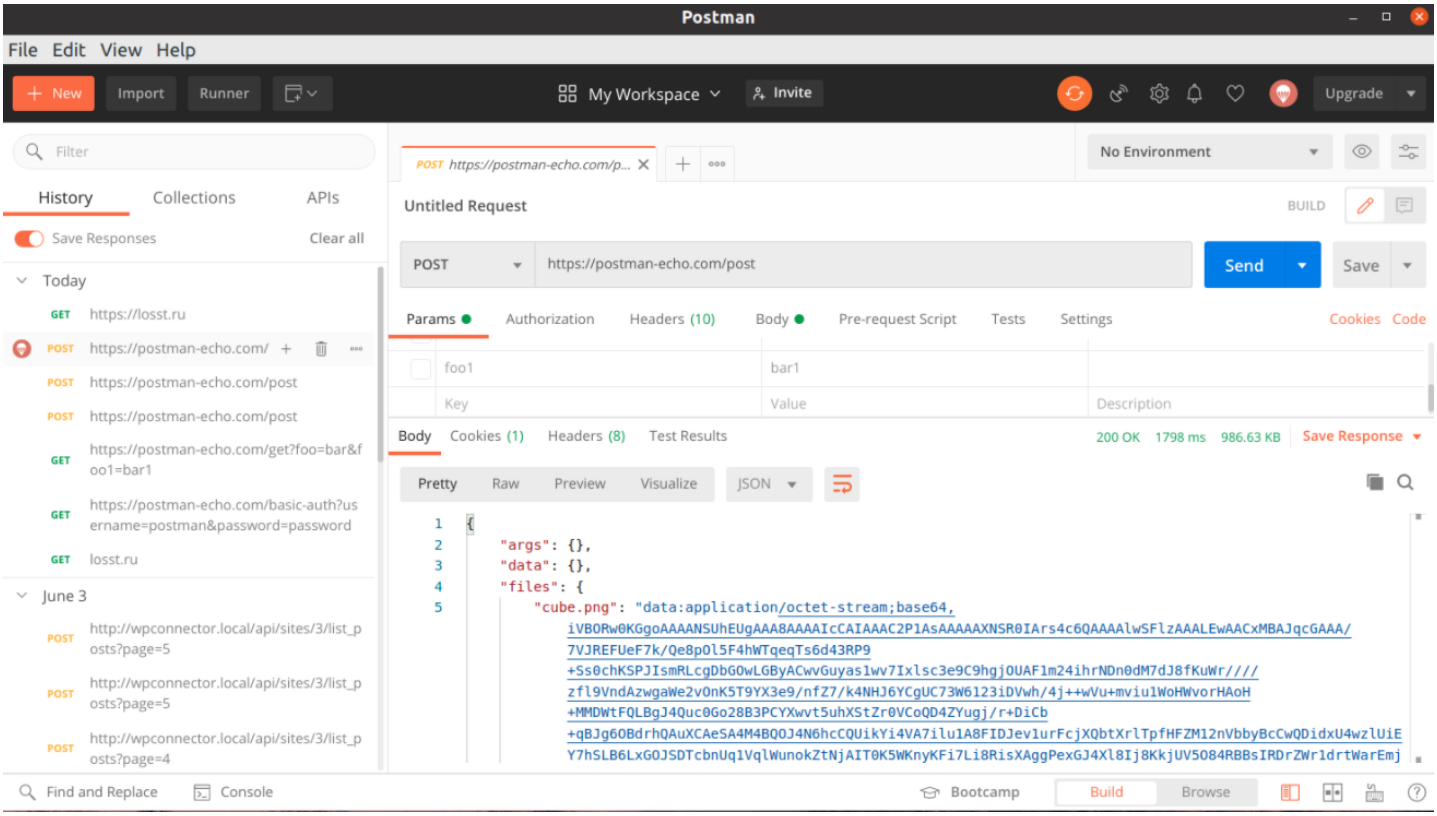
Если на сайте используется защита с авторизацией по методу Basic Auth, программа Postman дает возможность проверить ее прохождение. В качестве примера обращение будет осуществляться по адресу https://postman-echo.com/basic-auth. Чтобы пройти проверку, понадобится отправить значение имени пользователя **postman** и пароль доступа **password**.



Далее в рабочей области надо открыть вкладку **Authorization**, в разделе **Type** выбрать значение **Basic Auth** и заполнить имя с паролем. Если процедура пройдена успешно, тестовый сервер вернет ответ **authenticated: true**.

История и коллекция запросов:

При систематическом использовании одних и тех же запросов будет проще отправлять их из заранее составленного списка. Программа Postman упрощает задачу за счет сохранения истории, в которой содержатся все внесенные параметры. Пользователю лишь остается кликнуть по нужному пункту и при необходимости заменить URL.



Untitled Request

POST https://postman-echo.com/post

foo1 bar1

200 OK 1798 ms 986.63 KB

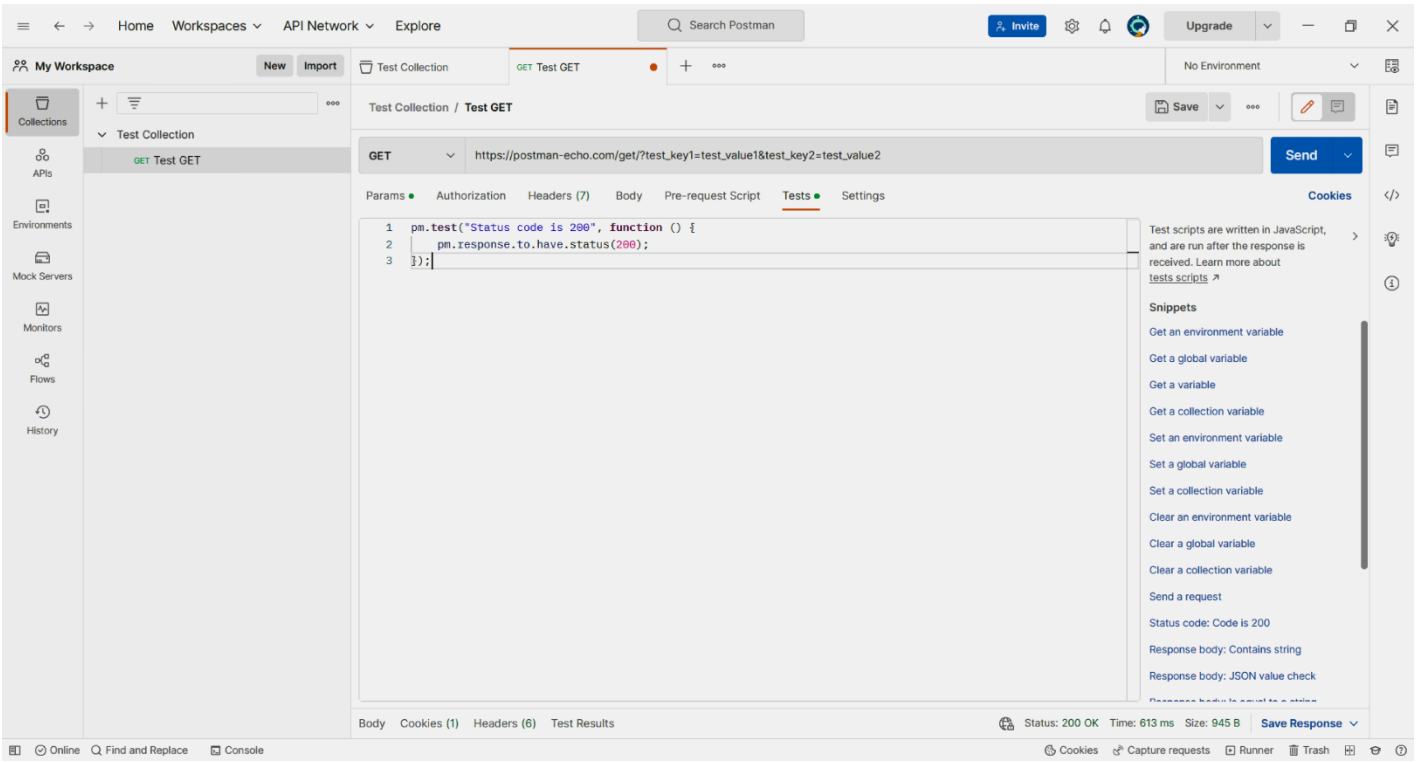
```
{
    "args": {},
    "data": {},
    "files": {
        "cube.png": "data:application/octet-stream; base64,
            iVBORw0KGgoAAAANSUhEUgAAA8AAAAIcCAIAAAC2P1AsAAAAAXNSR0IArs4c6QAAAAlwSFlzAAALEwAACxMBAJqcGAAA/
            7VJREFUeF7k/Qe8pOl5F4hWTqeqTs6d43RP9
            +Ss0chKSPJIsmRLcgDbGOwLGByACwvGuyas1wv7Ixlsc3e9C9hgjOUAF1m24ihrNDn0dM7dJ8fKuWr////
            zfl9VndAzwgaWe2vOnK5T9YX3e9/nfZ7/k4NHJ6YCgUC73W6123iDVwh/4j++wVu+mviu1WoHWvorHAoH
            +MMDWtFQLBgJ4Quc0Go28B3PCYXwvt5uhXStZr0VCoQD4ZYugj/r+DiCb
            +qBJg6OBdrhQAuXCAeSA4M4BQOJ4N6hcCQUikYi4VA7ilu1A8FIDJev1urFcjXQbtXrlTpfHFZM12nVbbyBcCwQDidxU4wzlUiE
            Y7hSLB6LxGOJSDTcbnUq1VqlWunokZtNjAIT0K5WKnyKFi7Li8RisXAggPexGJ4Xl8Ij8KkjUV5O84RBBsIRDrZWr1drtWarEmj
```

Наиболее важные запросы рекомендуется сохранять в коллекции. Чтобы сделать это, достаточно нажать на кнопку **New** на верхней панели, выбрать пункт **Collection** и ввести название (на выбор пользователя). Теперь любой запрос будет добавлен в перечень нажатием на кнопку **Create** и, после заполнения всех данных, кнопку **Save** (до отправки на сервер).

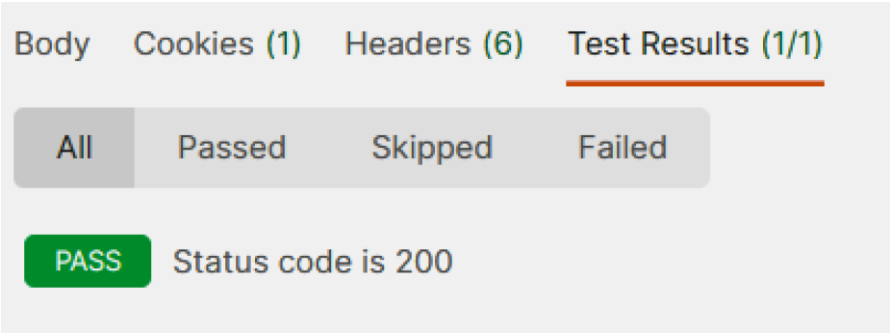
Как писать тесты:

Теперь напишем к нашему GET-запросу простой тест: проверку кода ответа. Если код ответа 200 (то есть запрос выполнен успешно) — тест пройден.

Для этого перейдём во вкладку **Tests** и в правой колонке найдём сниппет **Status code: Code is 200**. Нажмём на него — появится скрипт.

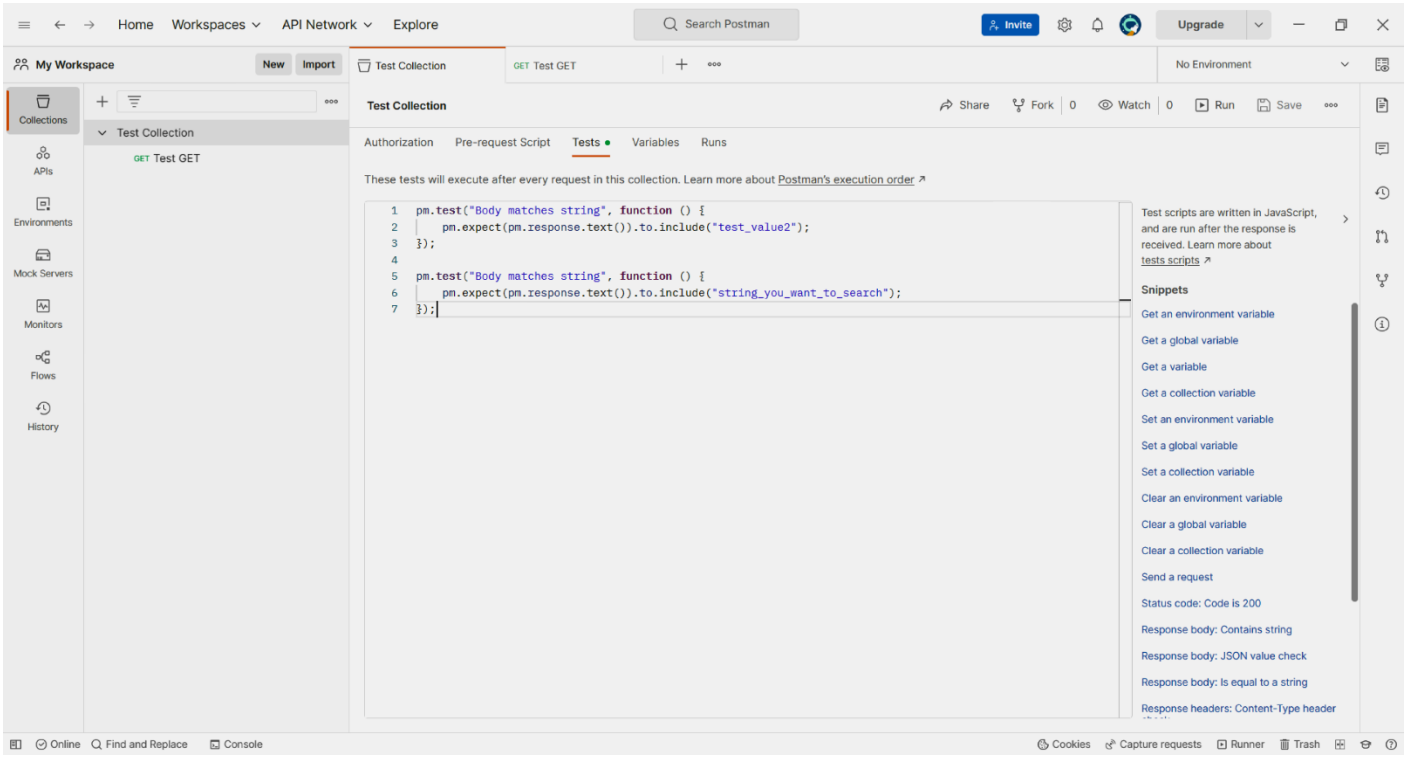


Теперь отправим запрос ещё раз. В поле ответа во вкладке Test Results появится отчёт о тесте: PASS (то есть пройден успешно).

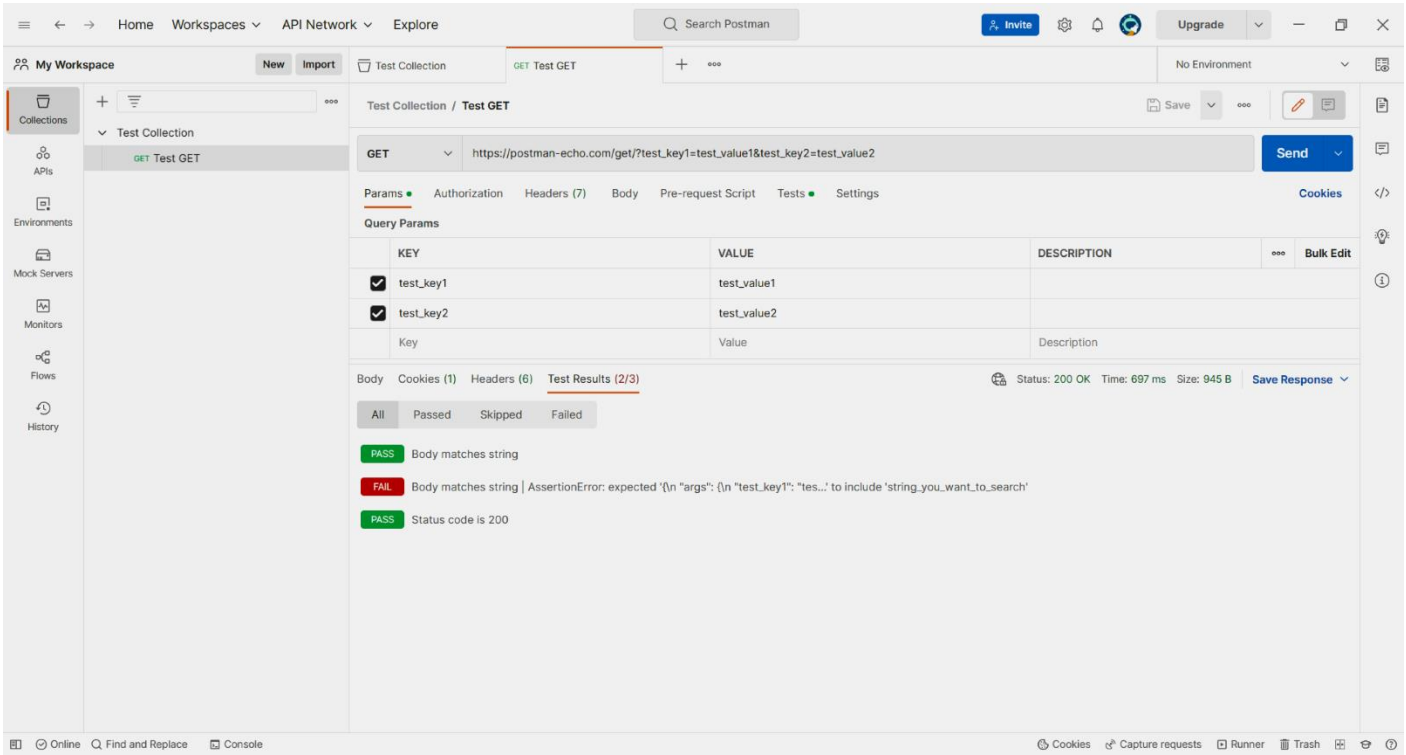


Теперь напишем ещё два теста, но не внутри отдельного запроса, а внутри коллекции. Первый тест будет проверять, есть ли в теле ответа строка test_value2, второй — есть ли там строка string_you_want_to_search.

Перейдём к **Test Collection**, на вкладку **Tests**. Выберем сниппет **Response body: Contains string** и нажмём на него два раза. В первом блоке кода строку string_you_want_to_search заменим на test_value2, во втором оставим как есть. Не забудем сохранить изменения, чтобы новые тесты применились к запросу.



Теперь отправим к эхо-серверу тот же GET-запрос. Он должен пройти первый тест (потому что мы передавали test_value2 в качестве одного из значений) и не пройти второй. Посмотрим на результат.



Всё как мы и предполагали. К запросу применились как его собственные тесты, так и тесты всей коллекции, один из которых он не прошёл. Причём тесты коллекции выполнились первыми — до тестов самого запроса.

Порядок выполнения скриптов, написанных в разных местах коллекции, таков:

- Сначала выполняются скрипты коллекции.
- Затем скрипты папки (там их тоже можно писать).
- И уже потом скрипты запроса.

Это работает с любым кодом: как внутри **Tests**, так и внутри **Pre-request Scripts**.

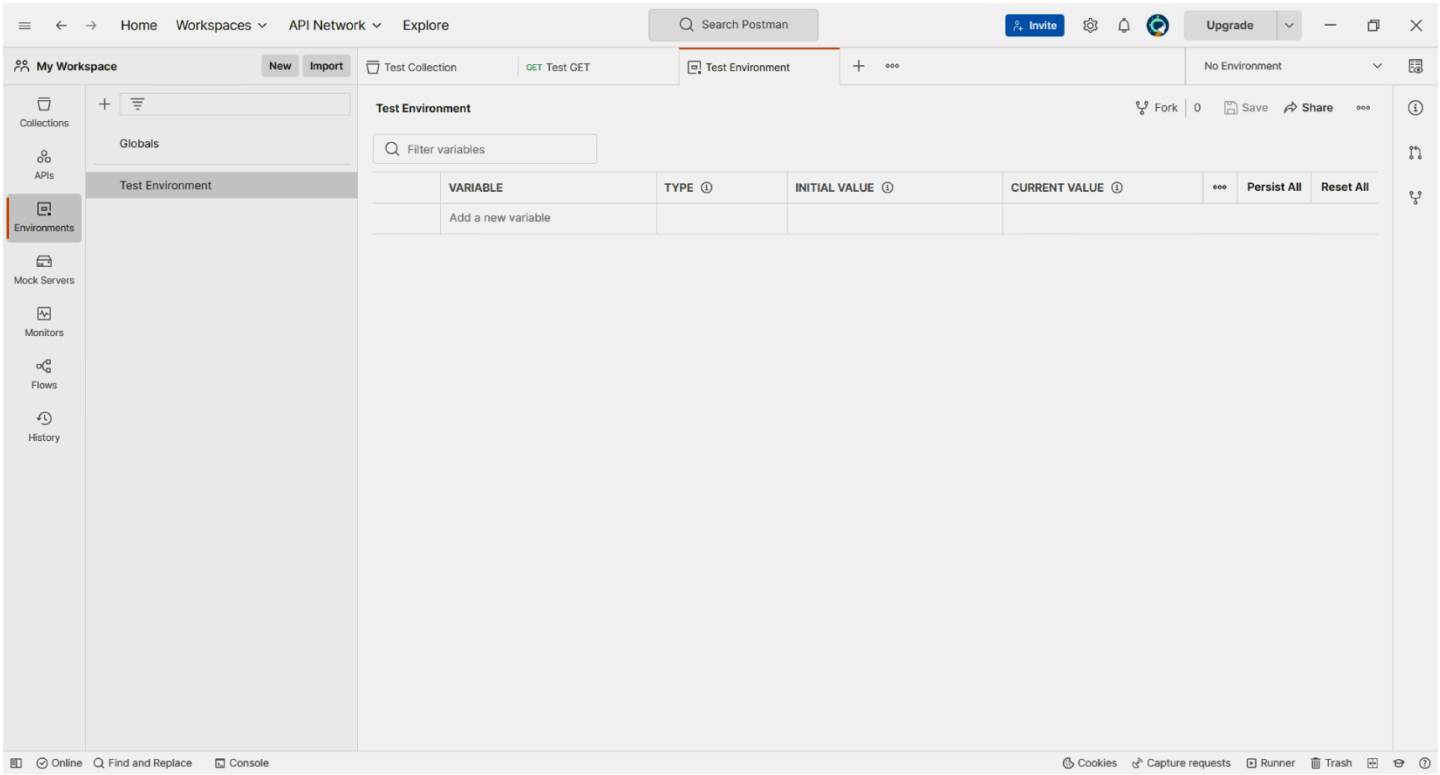
Как создать переменную:

Переменные, как и тесты, «живут» на разных уровнях Postman, и писать их можно в разных местах:

- **Глобальные переменные** применяются ко всему рабочему пространству. Написать их можно во вкладке **Environments** в разделе **Globals**.
- **Переменные коллекции** создаются внутри конкретной коллекции и работают только внутри неё.
- **Переменные окружения** задаются на уровне окружения во вкладке **Environments**. Чтобы применить их к запросу, нужно напрямую связать его с окружением.
- **Локальные переменные** существуют на уровне скриптов, которые выполняются при отправке запросов.
- **Переменные данных** возникают, когда мы пользуемся **Collection Runner** — инструментом для запуска сразу всех скриптов внутри коллекции или папки.

Чем меньше область видимости переменной, тем выше её приоритет. То есть если где-то встретятся глобальная и локальная переменная с одинаковыми именами, то применится значение локальной переменной.

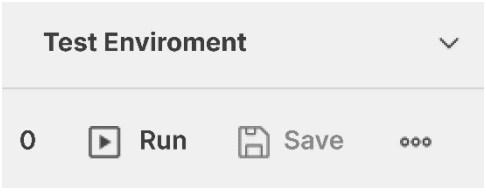
Создадим новое окружение **Test Environment** — сделать это можно по аналогии с созданием коллекции, но только во вкладке **Environment**.



Создадим переменную test_variable и присвоим ей значение test_value3. Сохраним изменения в окружении.

	VARIABLE	TYPE	INITIAL VALUE	CURRENT VALUE	Persist All	Reset All
☑	test_variable	default	test_value3	test_value3		
	Add a new variable					

Затем перейдём к тестовой коллекции и применим к ней окружение, выбрав его в правом верхнем углу. Теперь мы можем использовать в этой коллекции все переменные окружения.



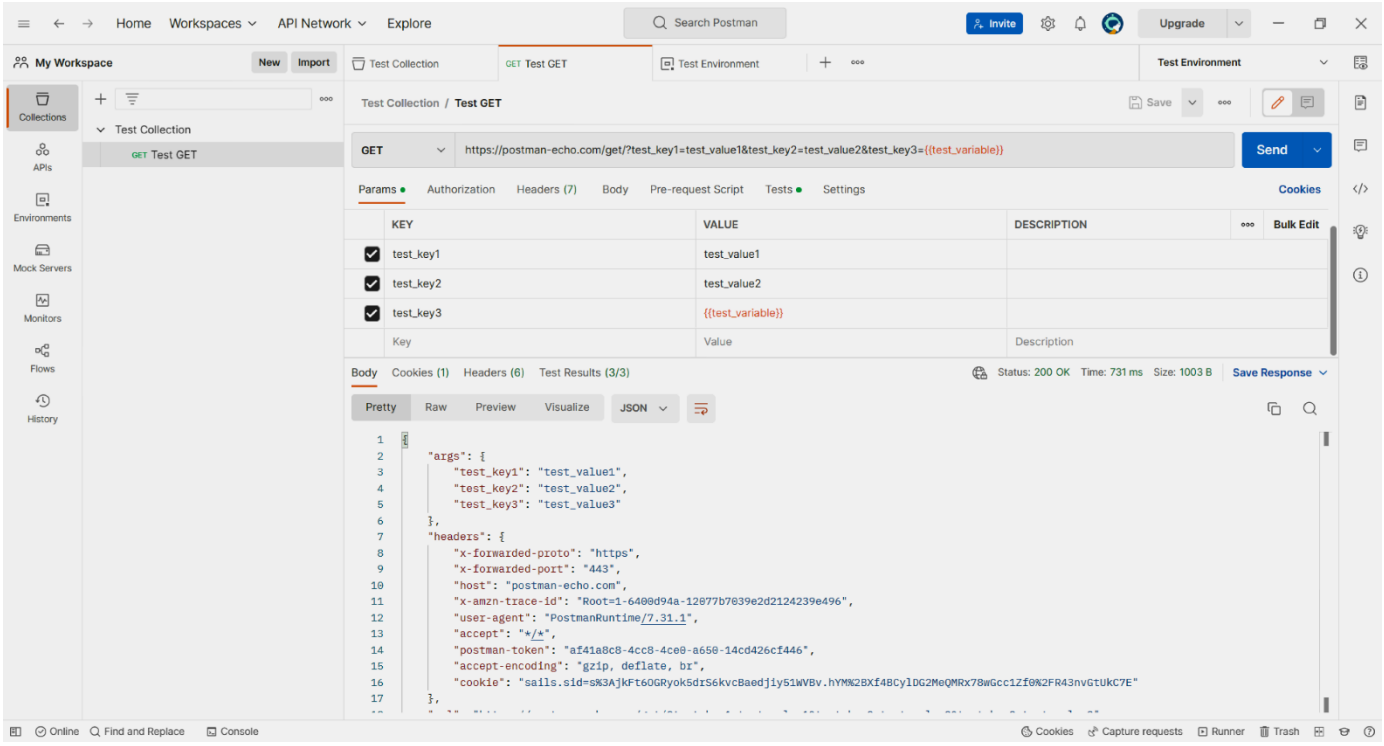
Усовершенствуем второй скрипт в тестах коллекции. Для этого строку "string_you_want_to_search" заменим на сниппет **Get an environment variable**. В нём variable_key заменим на название переменной test_variable. Сохраним изменения.

Таким образом, второй скрипт будет проверять, есть ли в ответе сервера соответствие не строке string_you_want_to_search, а содержимому переменной test_variable.

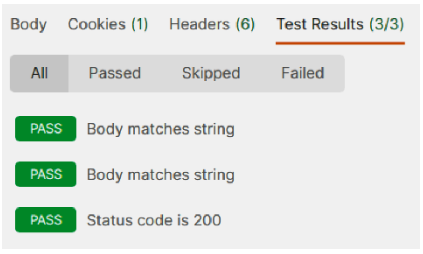
```
pm.test("Body matches string", function () {
    pm.expect(pm.response.text()).to.include("test_value2");
});

pm.test("Body matches string", function () {
    pm.expect(pm.response.text()).to.include(pm.environment.get("test_variable"));
});
```

Теперь перейдём к GET-запросу и создадим новый параметр. Его ключом будет test_key3, значением — содержимое переменной test_variable. Для этого её название заключим в двойные фигурные скобки {{}}. Сохраним изменения и отправим запрос.



Как видим, все тесты пройдены. Значит, значение нашей переменной корректно извлекается и в параметрах запроса, и в тестах.



SOAP – это протокол обмена сообщениями, который позволяет распределённым элементам приложения обмениваться данными (простой протокол доступа к объектам). SOAP может передаваться по множеству стандартных протоколов, он гибкий и независимый. Это позволяет разработчикам писать API на разных языках, а ещё добавлять различные функции и функциональные возможности.

SOAP работает с операциями.

Подход SOAP определяет, как будут обрабатываться сообщения, включенные функции и модули, поддерживаемые протоколы связи.

Информационный набор Extensible Markup Language **(XML*)** используется для SOAP в качестве формата сообщений, а передача и их согласование происходит с помощью протоколов прикладного уровня – таких, как HTTP.

Фишки SOAP:

- SOAP обладает высокой расширяемостью, но используется только те фрагменты, которые нужны для конкретной задачи.
- Частью волшебства является язык описания веб-служб (**WSDL**) – ещё один файл, связанный с SOAP. Он показывает, как работает веб-служба, поэтому при создании ссылки на нее среда IDE может полностью автоматизировать процесс.
- SOAP работает с XML, а другим языкам предоставляет ярлыки, которые могут ускорить и облегчить процесс создания запроса и анализа ответа.
- SOAP не терпит ошибок. И это не прикол! В некоторых языках программирования запросы и получение ответов в SOAP нужно будет создавать вручную, поэтому keep calm, please! Сложность зависит от языка программирования.

*** XML** (англ. eXtensible Markup Language) — расширяемый язык разметки, предназначенный для хранения и передачи данных.

Простейший XML-документ выглядит следующим образом:

```
<?xml version="1.0" encoding="windows-1251"?>
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price></price>
</book>
```

Первая строка — это XML декларация. Здесь определяется версия XML (1.0) и кодировка файла. На следующей строке описывается корневой элемент документа `<book>` (*открывающий тег*). Следующие 4 строки описывают дочерние элементы корневого элемента (`title`, `author`, `year`, `price`). Последняя строка определяет конец корневого элемента `</book>` (*закрывающий тег*).

Документ XML состоит из *элементов* (elements). Элемент начинается *открывающим тегом* (start-tag) в угловых скобках, затем идет *содержимое* (content) элемента, после него записывается *закрывающий тег* (end-teg) в угловых скобках.

Информация, заключенная между тегами, называется содержимым или значением элемента: `<author>Erik T. Ray</author>`. Т.е. элемент `author` принимает значение `Erik T. Ray`. Элементы могут вообще не принимать значения.

Элементы могут содержать *атрибуты*, так, например, открывающий тег `<title lang="en">` имеет атрибут `lang`, который принимает значение `en`. Значения атрибутов заключаются в кавычки (двойные или ординарные).

Некоторые элементы, не содержащие значений, допустимо записывать без закрывающего тега. В таком случае символ `/` ставится в конце открывающего тега:

```
<name first="Иван" second="Петрович" />
```

Структура XML:

XML документ должен содержать корневой элемент. Этот элемент является «родительским» для всех других элементов.

Все элементы в XML документе формируют иерархическое дерево. Это дерево начинается с корневого элемента и разветвляется на более низкие уровни элементов.

Все элементы могут иметь подэлементы (дочерние элементы):

```
<корневой>
  <потомок>
    <подпотомок>.....</подпотомок>
  </потомок>
</корневой>
```

Правила синтаксиса (Валидность)

Структура XML документа должна соответствовать определенным правилам. XML документ, отвечающий этим правилам называется *валидным* (*англ.* Valid — правильный) или *синтаксически верным*. Соответственно, если документ не отвечает правилам, он является *невалидным*.

Основные правила синтаксиса XML:

1. Теги XML регистрозависимы — теги XML являются регистрозависимыми. Так, тег `<Letter>` не то же самое, что тег `<letter>`.

Открывающий и закрывающий теги должны определяться в одном регистре:

```
<Message>Это неправильно</message>
<message>Это правильно</message>
```

2. XML элементы должны соблюдать корректную вложенность:

```
<b><i>Некорректная вложенность</b></i>
<b><i>Корректная вложенность</i></b>
```

3. У XML документа должен быть корневой элемент — XML документ должен содержать один элемент, который будет родительским для всех других элементов. Он называется корневым элементом.

Примечание

В большинстве XML файлов отчетов для ФНС корневым элементом является `<Файл></Файл>`. После закрывающего тега `</Файл>` больше ничего быть не должно.

4. Значения XML атрибутов должны заключаться в кавычки:

```
<note date="12/11/2007">Корректная запись</note>
<note date=12/11/2007>Некорректная запись</note>
```

Сущности:

Некоторые символы в XML имеют особые значения и являются служебными. Если вы поместите, например, символ `<` внутри XML элемента, то будет сгенерирована ошибка, так как парсер интерпретирует его, как начало нового элемента.

В примере ниже будет сгенерирована ошибка, так как в значении `"ООО<Мосавтогруз>"` атрибута `НаимОрг` содержатся символы `<` и `>`.

```
<НПЮЛ ИННЮЛ="7718962261" КПП="771801001" НаимОрг="ООО<Мосавтогруз>"/>
```

Также ошибка будет сгенерирована и в следующем примере, если название организации взять в обычные кавычки (английские двойные):

```
<НПЮЛ ИННЮЛ="7718962261" КПП="771801001" НаимОрг="ООО"Мосавтогруз""/>
```

Чтобы ошибки не возникали, нужно заменить символ `<` на его сущность. В XML существует 5 предопределенных сущностей:

Таблица I.1 — Сущности

Сущность	Символ	Значение
<	<	меньше, чем
>	>	больше, чем
&	&	амперсанд
'	'	апостроф
"	"	кавычки

Примечание

Только символы < и & строго запрещены в XML. Символ > допустим, но лучше его всегда заменять на сущность.

Таким образом, корректными будут следующие формы записей:

<НПЮЛ ИННЮЛ="7718962261" КПП="771801001" НаимОрг="ООО"Мосавтогруз""/>

или

<НПЮЛ ИННЮЛ="7718962261" КПП="771801001" НаимОрг="ООО«Мосавтогруз»"/>

В последнем примере английские двойные кавычки заменены на французские кавычки («ёлочки»), которые не являются служебными символами.

Поиск информации в XML файлах (XPath):

XPath (*англ.* XML Path Language) — язык запросов к элементам XML-документа. XPath расширяет возможности работы с XML.

XML имеет древовидную структуру. В документе всегда имеется корневой элемент (инструкция <?xml version=”1.0”?> к дереву отношения не имеет). У элемента дерева всегда существуют потомки и предки, кроме корневого элемента, у которого предков нет, а также тупиковых элементов (листьев дерева), у которых нет потомков. Каждый элемент дерева находится на определенном уровне вложенности (далее — «уровень»). У элементов на одном уровне бывают предыдущие и следующие элементы.

Это очень похоже на организацию каталогов в файловой системе, и строки XPath, фактически, — пути к «файлам» — элементам. Рассмотрим пример списка книг:

```
<?xml version="1.0" encoding="windows-1251"?>
<bookstore>
  <book category="COOKING">
    <title lang="it">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
```

```
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XPath запрос `/bookstore/book/price` вернет следующий результат:

```
<price>30.00</price>
<price>29.99</price>
<price>39.95</price>
```

Сокращенная форма этого запроса выглядит так: `//price`.

С помощью XPath запросов можно искать информацию по атрибутам. Например, можно найти информацию о книге на итальянском языке: `//title[@lang="it"]` вернет `<title lang="it">Everyday Italian</title>`.

Чтобы получить больше информации, необходимо модифицировать запрос `//book[title[@lang="it"]]` вернет:

```
<book category="COOKING">
    <title lang="it">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
```

В приведенной ниже таблице представлены некоторые выражения XPath и результат их работы:

Таблица I.2 — Выражения XPath

Выражение XPath	Результат
`/bookstore/book[1]`	Выбирает первый элемент `book`, который является потомком элемента `bookstore`
`/bookstore/book[position()<3]`	Выбирает первые два элемента `book`, которые являются потомками элемента `bookstore`
`//title[@lang]`	Выбирает все элементы `title` с атрибутом `lang`
`//title[@lang='en']`	Выбирает все элементы `title` с атрибутом `lang`, который имеет значение `en`
`/bookstore/book[price>35.00]`	Выбирает все элементы `book`, которые являются потомками элемента bookstore и которые содержать элемент `price` со значением больше `35.00`

Таблица I.2 — Выражения XPath

Выражение XPath	Результат
/bookstore/book[price>35.00]/title	Выбирает все элементы title элементов book элементов bookstore, которые содержать элемент price со значением больше 35.00

Кодировки:

Самыми распространенными кириллическими кодировками являются Windows-1251 и UTF-8. Последняя является одним из стандартов, но большая часть ФНС отчетности имеет кодировку Windows-1251.

В XML файле кодировка объявляется в декларации:

<?xml version="1.0" encoding="windows-1251"?>

Часто можно столкнуться с ситуацией, когда текстовый редактор некорректно распознает кодировку и отображает кракозябры. В такой случае, необходимо выбрать кодировку вручную, для этого выполните:

Таблица I.3 — Смена кодировки в разных программах

Программа	Кодировка
Notepad++	*«Документ → Кодировка»*
Geany	*«Документ → Установить кодировку»*
Firefox	*«Вид → Кодировка»*
Chrome	*«Настройка → Дополнительные инструменты → Кодировка»*

Примечание

В большинстве случаев при работе с русскоязычными файлами помогает переключение кодировки на Windows-1251 или UTF-8. Если все равно не удается прочитать содержимое XML документа, стоит открыть его в Mozilla Firefox, он отлично распознает кодировки.

Если ничего не помогает, вполне возможно, что файл был поврежден.

XSD схема:

XML Schema — язык описания структуры XML-документа, его также называют **XSD**. Как большинство языков описания XML, XML Schema была задумана для определения правил, которым должен подчиняться документ. Но, в отличие от других языков, XML Schema была разработана так, чтобы её можно было использовать в создании программного обеспечения для обработки документов XML.

После проверки документа на соответствие XML Schema читающая программа может создать модель данных документа, которая включает:

- словарь (названия элементов и атрибутов);
- модель содержания (отношения между элементами и атрибутами и их структура);
- типы данных.

Каждый элемент в этой модели ассоциируется с определённым типом данных, позволяя строить в памяти объект, соответствующий структуре XML-документа. Языкам

объектно-ориентированного программирования гораздо легче иметь дело с таким объектом, чем с текстовым файлом.

WSDL:

WSDL - это описательный язык, основанный на языке разметки XML, и именно в wsdl описан веб-сервис, который вам придется тестировать. (Можно сказать, что это полная документация веб-сервиса). WSDL включает в себя информацию о местоположении сервиса, часто включает в себя XSD. Именно из WSDL SOAPUI генерирует проверяемые классы.

Если Вы направите в веб-сервис нестандартный запрос, он ответит на это ошибкой. WSDL - это свод правил общения с вашим сервисом, соблюдая которые вы сможете с этим сервисом коммуницировать. Собственно, WSDL и XSD подробно описывают что и в каком виде слать на сервер, чтобы получить хороший ответ.

Основные теги, с которыми вы столкнетесь в описании WSDL-сервера:

- **Message** — сообщения, используемые web-сервисом.
- **PortType** — список операций, которые могут быть выполнены с сообщениями.
- **Binding** — способ, которым сообщение будет доставлено.

Для работы с SOAP API используют Soap UI:

SoapUI - это приложение для тестирования веб-сервисов с открытым исходным кодом для сервисно-ориентированных архитектур (SOA) и передач состояния представления (REST).

Язык программирования, на котором пишут скрипты в SOAP UI, называется Groovy.

Его функциональность охватывает проверку веб-сервисов, вызов, разработку, моделирование и имитацию, функциональное тестирование, нагрузочное тестирование и проверку соответствия.

Он полностью построен на платформе Java и использует Swing для пользовательского интерфейса. Это означает, что SoapUI является кросс-платформенным. Сегодня SoapUI также поддерживает IDEA, Eclipse и NetBeans.

SoapUI может тестировать веб-службы SOAP и REST, JMS, AMF, а также совершать любые вызовы HTTP(S) и JDBC.

Как отправлять запросы:

New REST Project → Пишем URI → Запрос создаётся, можно добавлять новые.

Method можно выбирать из выпадающего списка.

Проект имеет иерархическую структуру.

Project → Service → Resource → Method → Request

Названия отражают суть:

Request (запрос) это то место, где можно поменять тело запроса и просмотреть ответ сервера. Чтобы отправлять запросы нужно нажимать зелёный треугольник.

Method (метод) указывает GET, POST, PUT или другой метод, который Вы будете использовать. Все дочерние запросы будут иметь один и тот же Method.

Resource (ресурс) отвечает за ту часть URL, которая добавляется к базовой.

Service (сервис) отвечает за базовую часть URL

Как сохранять проекты:

Советую помимо использования «Save all projects» закрывать все новые окна вручную. Тогда SOAP UI предложит Вам сохранить каждый проект по отдельности.

После того, как все новые проекты сохранены, Вы можете закрыть SOAP UI. Я стараюсь закрывать SOAP UI только когда все окна закрыты.

Создание Test Suit in SOAP UI:

Коротко:

→Правый клик Project → Create New TestSuit (CTRL + T)

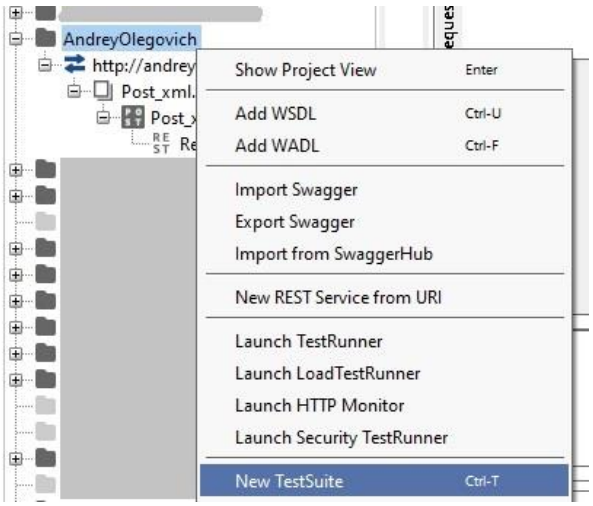
Укажите имя для TestSuit → Правый клик на TestSuit

New TestCase (CTRL + N) → Укажите имя для TestCase
Expand Test Case → Кликните на Test Steps → Add Step
Выберите request (e.g. SOAP Request) → Укажите имя для new step
Выберите операцию, которая запускает request
Добавьте Request в TestCase (OK)
Зелёная «+» иконка плюса появится наверху.
Кликнув на неё Вы можете добавить Assertions.

Подробно:

Правый клик на названии проекта

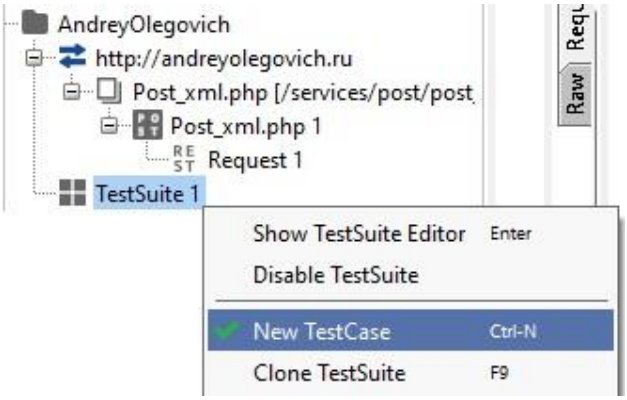
→ Create New TestSuit (CTRL + T)



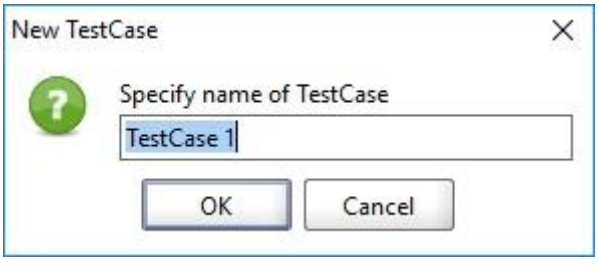
Укажите имя для TestSuit

→ Правый клик на TestSuit

New TestCase (CTRL + N)



Укажите имя для TestCase



→ Expand Test Case
→ Кликните на Test Steps
→ Add Step
→ Выберите request (e.g. SOAP Request)
→ Укажите имя для new step
→ Выберите операцию, которая запускает request
→ Добавьте Request в TestCase (OK)

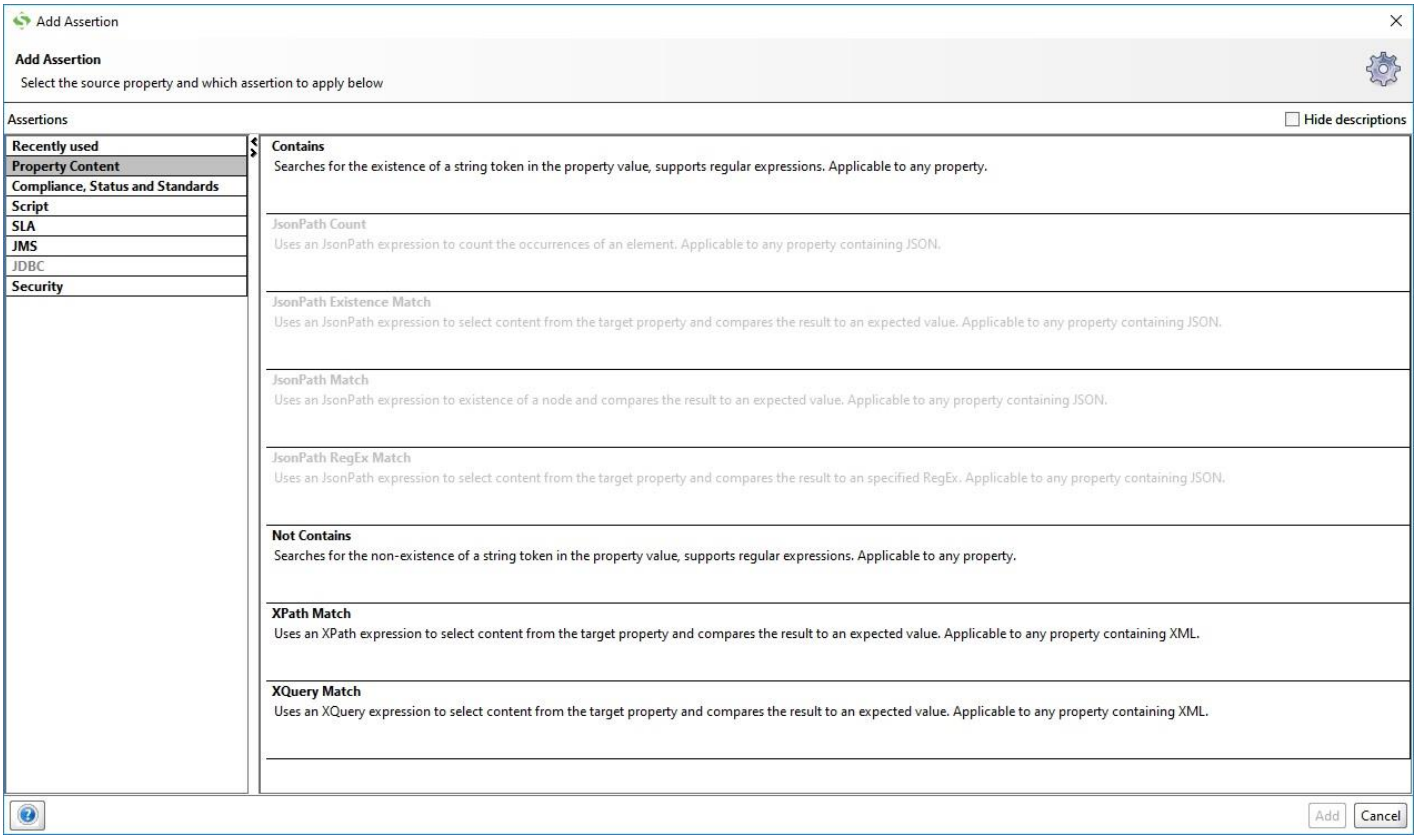
Зелёная «+» иконка плюса появится наверху. Кликнув на неё Вы можете добавить Assertions.

Assertions в SOAP UI:

Assertions добавляются в TestSuit

Поэтому, чтобы добавлять Assertions нужно создать хотя бы один TestSuit и затем кликнуть на зелёную+ иконку.

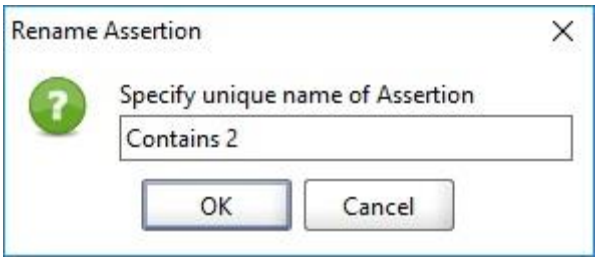
Затем Вы можете выбрать один из многих доступных в SOAP UI типов Assertions.



Property Content → Contains:

Выберите **Contains**. С помощью этого подтверждения (Assertions) можно искать присутствует ли в ответе заранее определённое ключевое слово. Оно поддерживает регулярные выражения и применимо ко всем ответам.

Specify unique name of Assertion



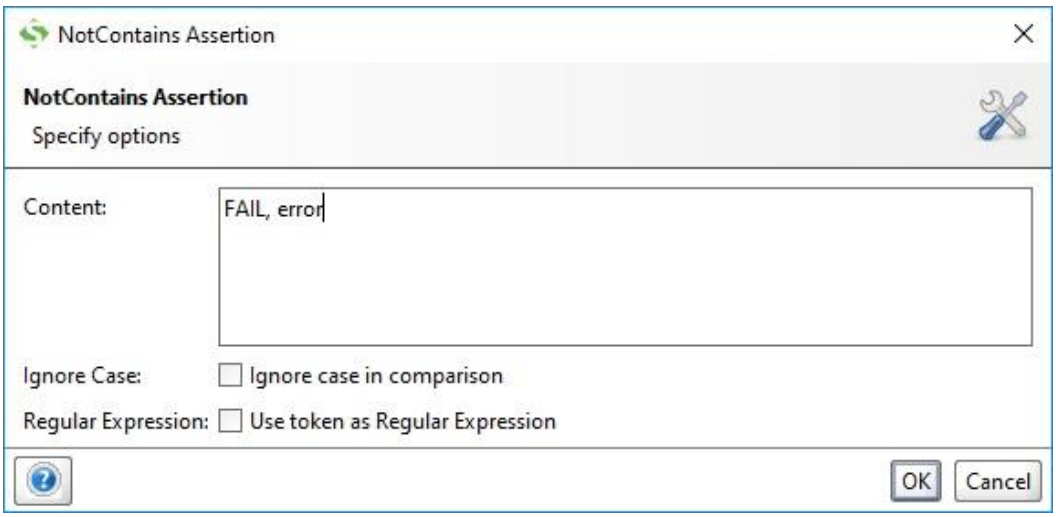
Type in content that you expect to receive in case of successful request

Property Content → Not Contains:

Выберите **Not Contains**. С помощью этого подтверждения (Assertions) можно проверить отсутствие в ответе заранее определённого ключевого слова. Оно поддерживает регулярные выражения и применимо ко всем ответам.

Введите уникальное имя для Assertions

Введите сюда то, что Вы точно не хотите видеть в теле ответа

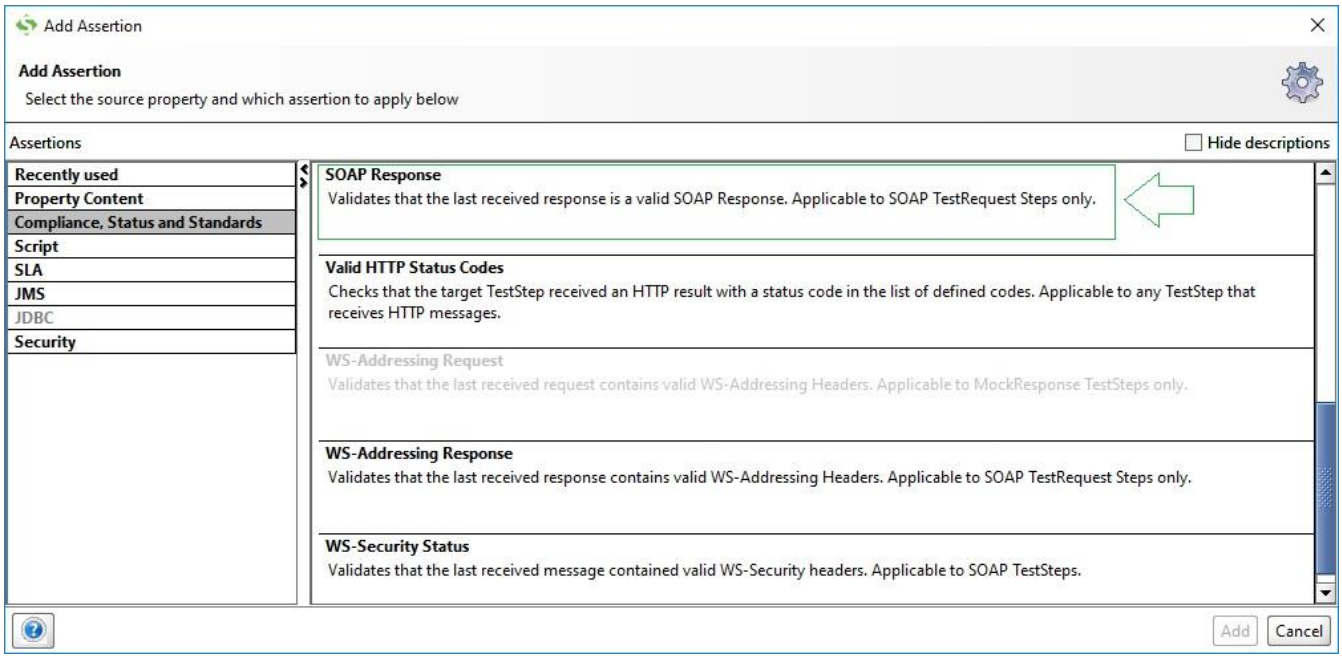


Compliance, Status и Standards:

SOAP Response:

Выберите **SOAP Response**. Этим assertion вы можете проверить что последний полученные ответ является валидным SOAP ответом. Это можно применять только к SOAP TestRequest Steps. Не пытайтесь применять этот assertion к REST запросам.

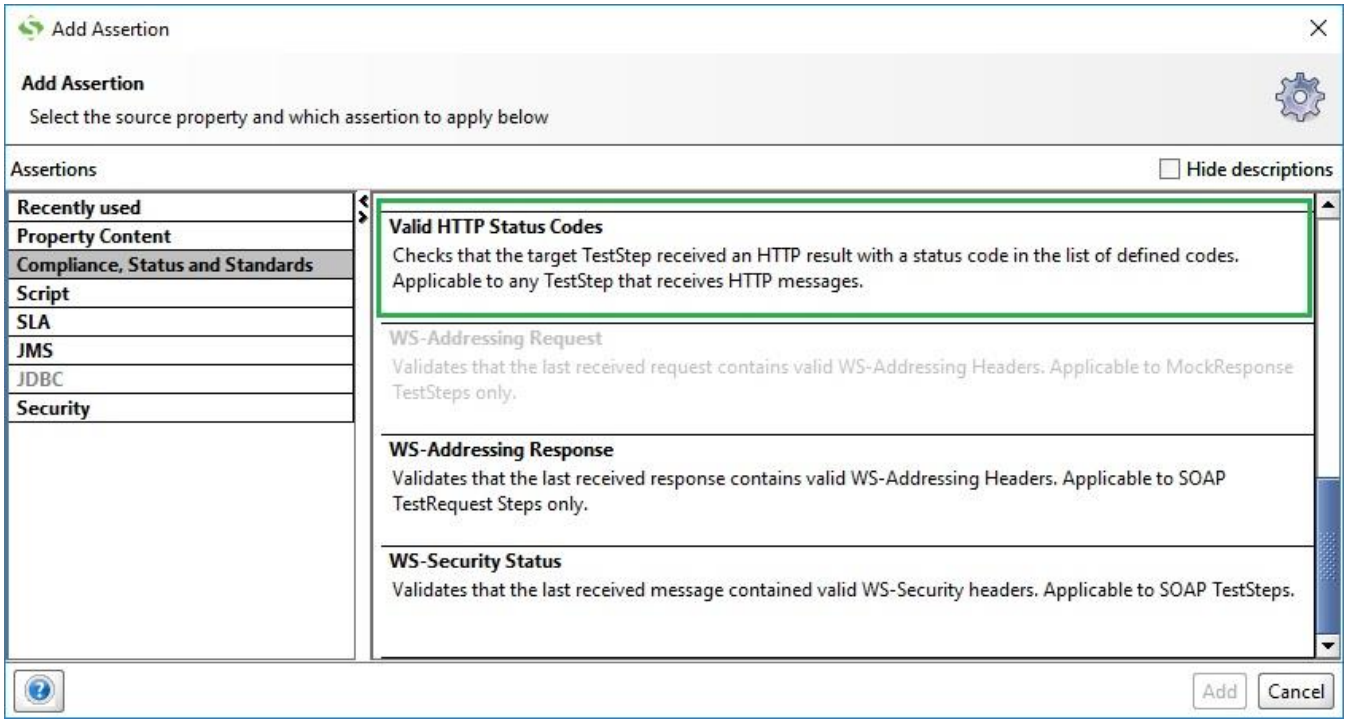
Двойной клик на Assertion. Никакие дополнительные параметры не нужны этот шаг можно добавить только один раз.



Compliance, Status и Standards:

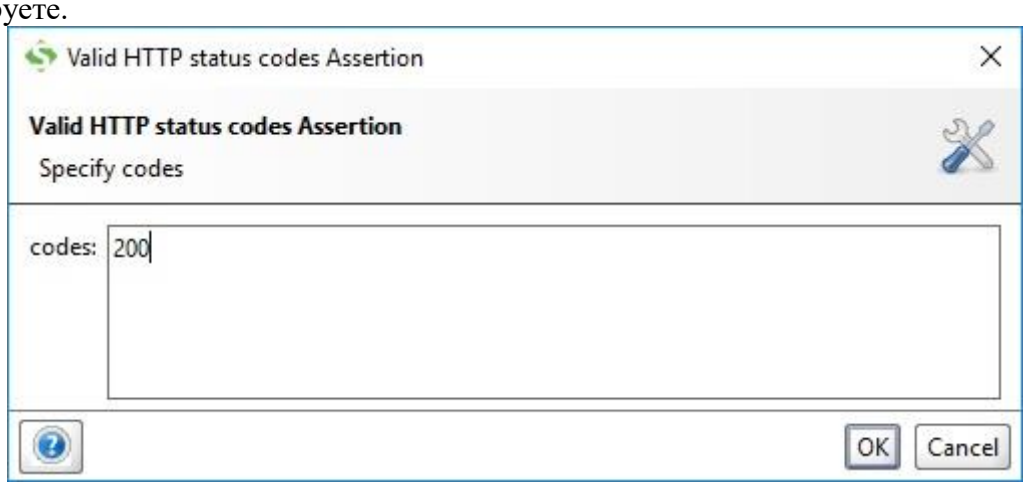
Valid HTTP Status Code:

Выберите **Valid HTTP Status Code**. С помощью этого assertion Вы можете проверить является ли последний полученный ответ (Response) валидным SOAP ответом (Response). Как Вы уже, наверное, догадались, этот assertion применим только к SOAP TestRequest Steps. Не пытайтесь использовать его для REST запросов.



Type in the HTTP Status Codes that are appropriate for your request.

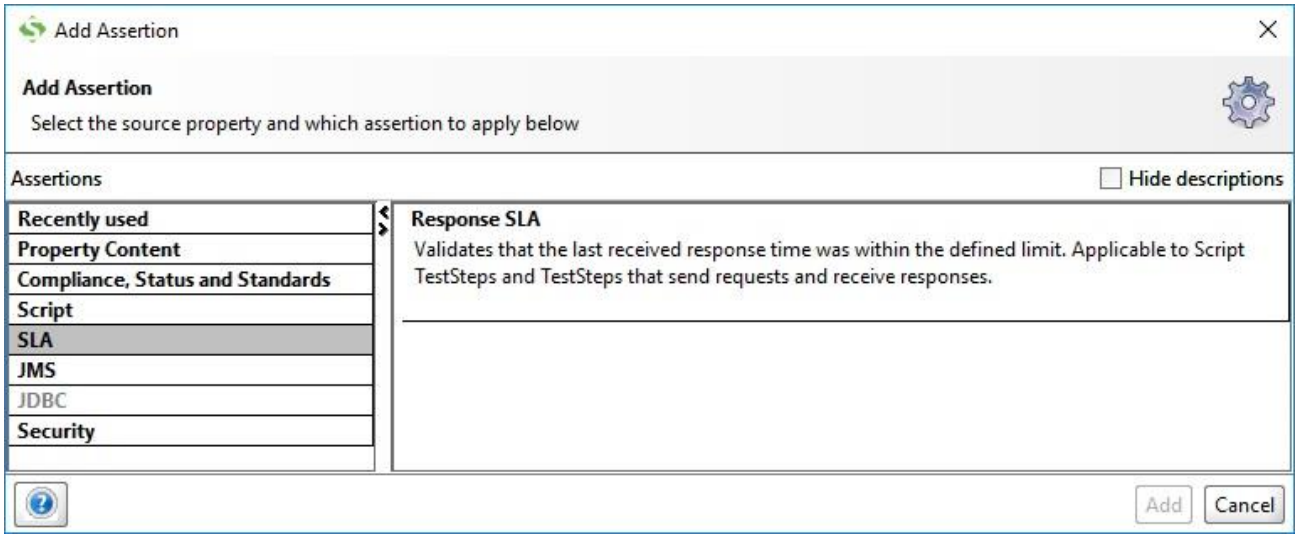
Обычно это 200, но всё же стоит прочитать спецификацию системы, которую Вы тестируете.



SLA → Response SLA:

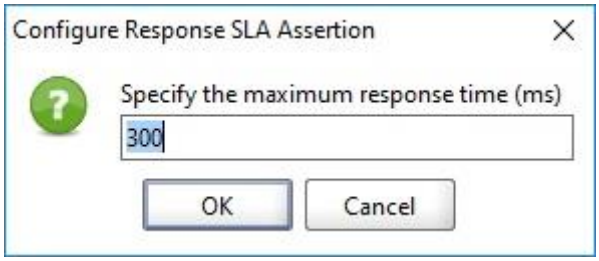
Выберите **Response SLA**. С помощью этого assertion Вы можете подтвердить, что последний полученный ответ (Response) был получен в течении определенного времени.

Можно применять к Script TestSteps и TestSteps которые посылают запросы и получают ответы.



Укажите максимальное время ответа (мс)

Если время, в которое нужно уложиться не указано в спецификации - поставьте какое-то разумное время.



Property Content → XPath Match:

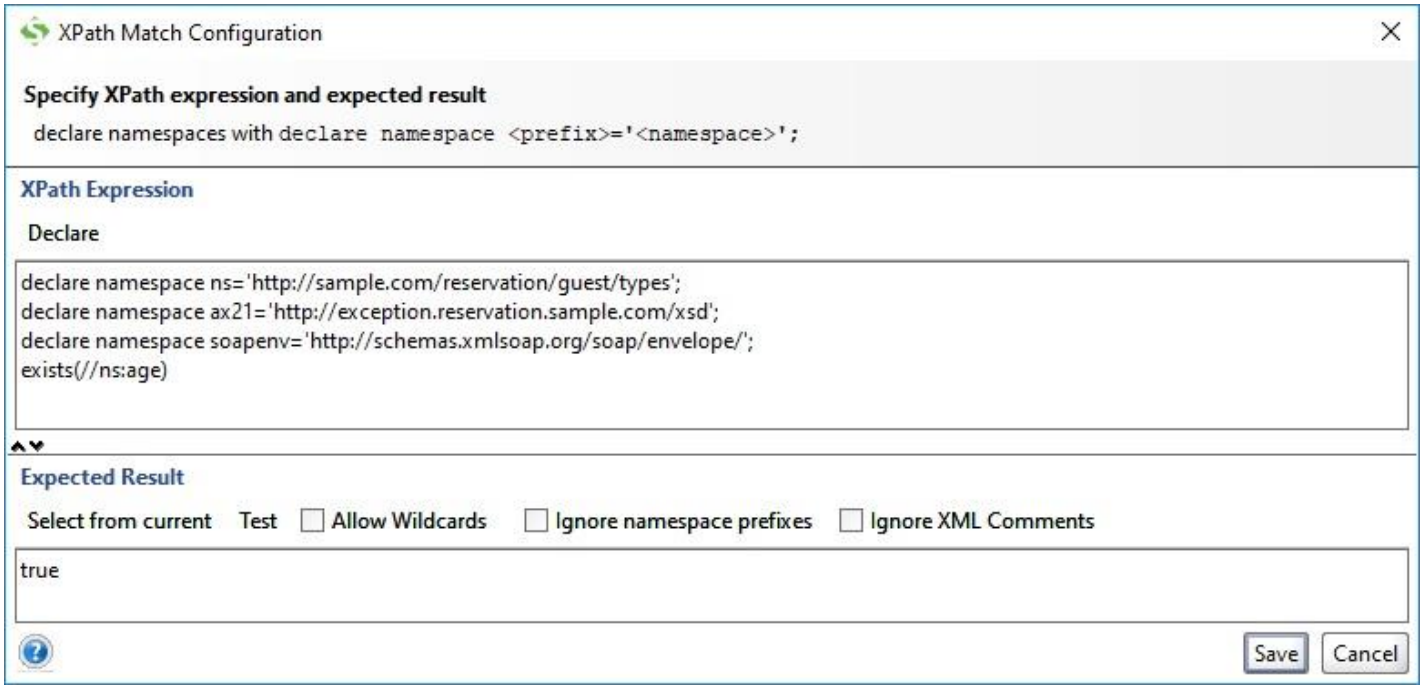
Выберите **XPath Match**. Этот Assertion использует выражение типа XPath чтобы выбрать содержимое указанного объекта и сравнить результат с ожидаемым значением.

Может быть применён к любому объекту, который содержит XML.

Нажмите Declare

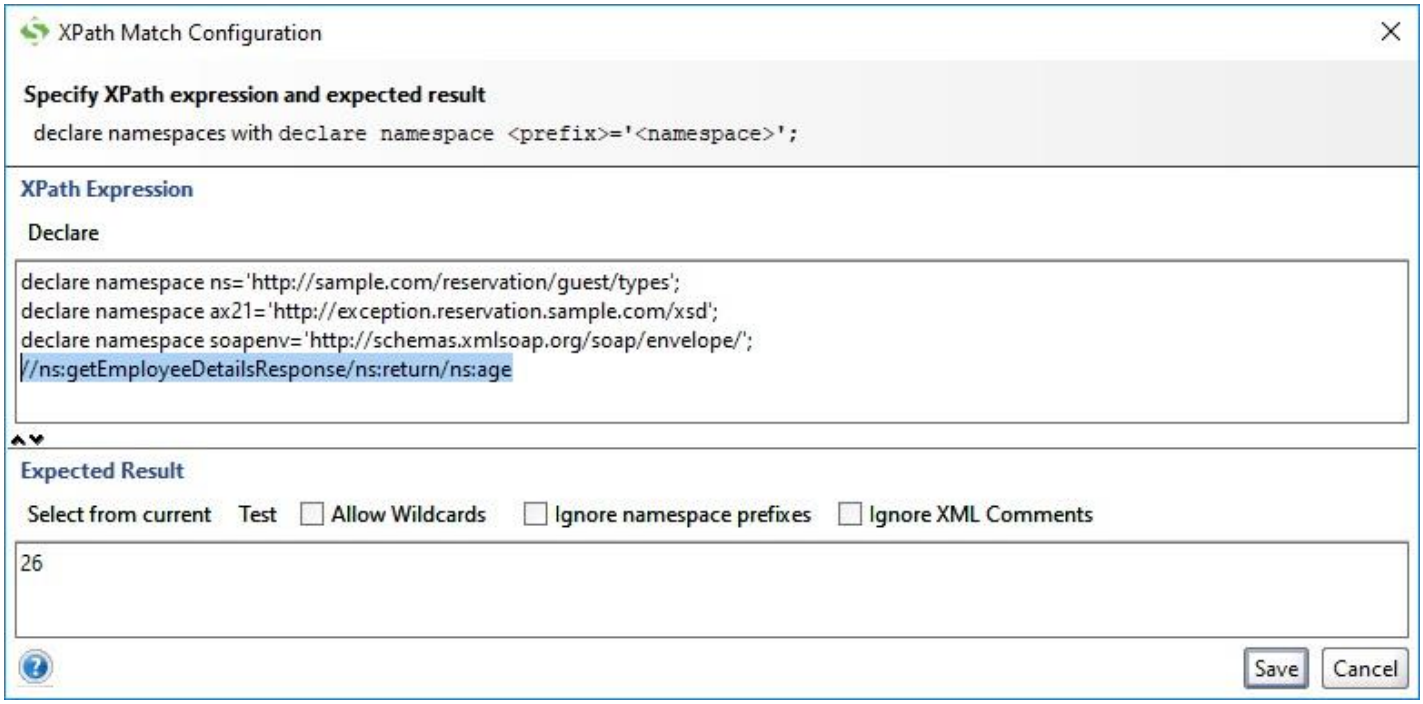
Чтобы подтвердить, что нужные данные присутствуют в ответе напишите **exists (//данные_которые_должны_придти).**

В поле Expected Result напишите **true**.

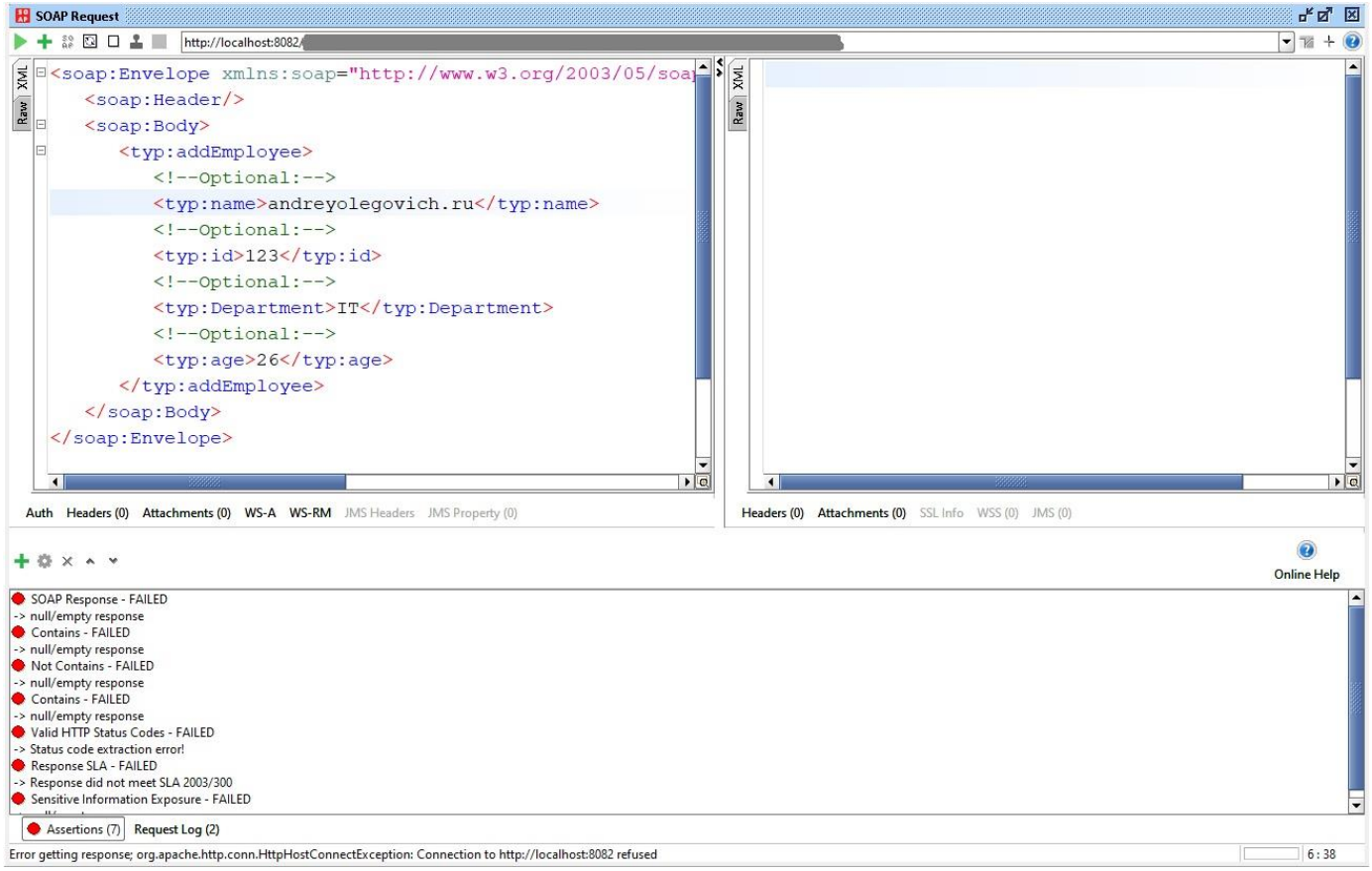


Чтобы проверить не наличие как таковое, а значение какой-то величины введите в **XPath Expression** либо полный путь до величины, либо её имя.

А само значение, которое Вы ожидаете получить введите в поле **Expected Result**.



В случае, если, например, мы не можем соединиться с сервером и все наши Assertions зафейлились у них появятся красные индикаторы.



Выводы:

REST (передача самоописываемого состояния) и SOAP (простой протокол доступа к объектам) предлагают разные методы вызова веб-службы. И если REST – это, как мы уже разобрали, архитектурный стиль, то SOAP определяет стандартную спецификацию протокола связи для обмена сообщениями на основе XML.

Веб-службы RESTful не имеют состояния. Реализация на основе REST проста по сравнению с SOAP, но пользователи должны понимать контекст и передаваемое содержимое, поскольку не существует стандартного набора правил для описания интерфейса веб-служб REST.

Службы REST полезны для устройств с ограниченным профилем (таких, как мобильные) и их легко интегрировать с существующими веб-сайтами.

Для SOAP требуется меньше кода (низкоуровневого инфраструктурного кода, который соединяет основные модули кода вместе), чем для проектирования служб REST. Язык описания веб-служб описывает общий набор правил для определения сообщений, привязок, операций и местоположения службы. Веб-службы SOAP полезны для асинхронной обработки и вызова.

SOAP и применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD, а вот в тех приложениях, которые не покидают рамки данной теории, вполне применимым может оказаться именно REST ввиду своей простоты и прозрачности. Действительно, если любым объектам вашего сервиса не нужны более сложные взаимоотношения, кроме: «Создать», «Прочитать», «Изменить», «Удалить» (как правило — в 99% случаев этого достаточно), возможно, именно REST станет правильным выбором. Кроме того, REST по сравнению с SOAP, может оказаться и более производительным, так как не требует затрат на разбор сложных XML команд на сервере (выполняются обычные HTTP запросы — PUT, GET, POST, DELETE). Хотя SOAP, в свою очередь, более надежен и безопасен.

gRPC:

RPC — это протокол вызова удаленных процедур, который одна программа может использовать, чтобы вызвать метод, функцию или процедуру в другой, доступной в сети. При этом не нужно досконально разбираться в сети. У клиента и сервера есть дополнительный уровень в общении — клиентская и серверная заглушки. Они созданы для того, чтобы отправлять данные по нужному адресу. Соответственно, в самом коде мы об этом вообще не переживаем. Эти заглушки генерируются автоматически. Все, что нам остается сделать — вызвать нужную процедуру.

Новая и, пожалуй, наиболее действенная и перспективная реализация этой концепции — фреймворк **gRPC**. По сравнению с другими реализациями RPC, у него есть множество преимуществ:

- **Идиоматические клиентские библиотеки на более чем 10 языках**. Мы можем использовать библиотеки на Java, C#, JavaScript, Python и т.д. Все это выглядит нативно — не так, словно мы используем какую-то инопланетную технологию.
- **Простая структура определения сервисов**. Для этого используются .proto-файлы.
- **HTTP/2**. Двунаправленная потоковая передача на основе HTTP/2.
- **Трассировка**. Позволяет мониторить вызовы процедур, что очень полезно для отладки приложения и анализа работы сервисов для их дальнейших оптимизаций и улучшений.
- **Health check**. С помощью этого механизма можно быстро проверить, работоспособен ли сервис и готов ли он обрабатывать запросы. Это помогает для балансировки нагрузки.
- **Балансировка нагрузки**. Эта особенность позволяет распределять нагрузку на несколько экземпляров серверного приложения, тем самым значительно упрощая вопрос масштабирования. Балансировка может производиться как на клиентской части (например, клиентское приложение поочередно отправляет запросы на разные серверы), так и на промежуточной (специальной прокси) посредством сервис-меша.
- **Подключение аутентификации**. Ее отсутствие было главным недостатком прошлых реализаций протокола. Из-за такой уязвимости RPC рекомендовали только для внутреннего общения.

Сравнивая гугловскую технологию с REST, здесь тоже находим свои плюсы:

- **Работа с Protobuf**. В REST для передачи данных применяется текстовый формат JSON, который не сжимается. Protobuf — это бинарный формат. Используя его, мы избегаем передачи лишних данных и нам не надо будет десериализовать после этого полученные сообщения.
- **Обработка HTTP-запросов**. В случае с REST необходимо постоянно думать, какой статус-код может прийти, какие данные будут храниться и т.п. В gRPC мы прикладываем минимум усилий для вызова удаленных процедур и их определения.
- **Простота определения контрактов**. В REST для описания интерфейсов и документации нужно использовать сторонние инструменты и библиотеки — такие, как OpenAPI или Swagger. В gRPC происходит простое определение контрактов в .proto-файлах.
- **HTTP/2**. REST зачастую использует более старую версию данного протокола — HTTP/1.1.

Чем хорош HTTP/2? Среди важных преимуществ:

- бинарный формат передачи данных (уменьшает размер сообщений и ускоряет работу);
- экономия трафика (усовершенствованное сжатие HTTP-сообщений, в первую очередь хедеров);
- возможность передавать потоки данных;

- мультиплексирование (в HTTP 1.1 для передачи трех файлов надо установить три соединения, в каждом из которых будет запрашиваться и отправляться определенный файл. В HTTP/2 можно все передать по одному соединению);
- приоритезация потоков.

Как настроить соединение по gRPC:

Последовательность создания gRPC канала включает несколько этапов:

1. Открытие сокетов.
2. Установка TCP-соединения.
3. Согласование TLS.
4. Запуск HTTP/2-соединения.
5. Выполнение вызова gRPC-процедуры.

Мы могли бы избежать первых четырех пунктов за счет того, что один раз устанавливаем соединение и просто пользуемся им — это называется Persistent Connection. Почему бы не работать с таким решением постоянно? Однако возникает вопрос, как настроить балансировку нагрузки, когда, допустим, нам понадобится несколько экземпляров сервиса — как их распределять? Вариантов несколько:

- **Балансировка нагрузки на стороне клиента**. В этом случае клиентская библиотека знает о существовании нескольких инстансов данного сервиса. Например, она будет отправлять запросы на каждый из них в процентном соотношении.
- **Балансировка нагрузки через прокси**. Если эти сервисы хостятся на каком-то оркестраторе (типа Kubernetes), он может решить, что инстансов слишком много, и убирает один или, наоборот, добавляет новый. В этом случае помогает балансировка нагрузки через прокси Service Mesh. Это может быть Linkerd, Istio, Consul и т.п. Клиент будет устанавливать одно постоянное соединение к Mesh, а уже он посмотрит, какие есть экземпляры сервиса, когда они появляются или исчезают и будет хендлить это. Соединение будет только к актуальным сервисам, а клиент об этом знать не будет — у него всегда один коннекшн.

Иногда gRPC сравнивают с WCF. Я не думаю, что это актуально, поскольку gRPC — это узконаправленный фреймворк, который хорошо решает одну задачу. WCF — более универсальный фреймворк, который поддерживает RPC, но также поддерживает REST, SOAP и т.п. К сожалению, WCF не является универсальным в плане поддерживаемых платформ, ведь пока он сильно привязан к .NET. В свою очередь gRPC может работать в любой среде, и писать его можно на любых языках из списка поддерживаемых.

Однако на данный момент gRPC не может полноценно функционировать в браузерах. Реализовать HTTP/2-общение в браузере невозможно, потому что нет такого контроля над каналом связи, какой может быть, допустим, в .NET-приложении. Поэтому Google предлагает альтернативу: использовать gRPC-прокси. То есть сам браузер будет отправлять запросы HTTP 1.1 на прокси, который будет мапить сообщение в вызов gRPC процедуры.

HTML/CSS:

HTML — это язык разметки гипертекстовых документов. Он нужен, чтобы отображать в браузере специальным образом отформатированный документ с множеством вложенных элементов: заголовками, абзацами, списками, гиперссылками, медиаисточниками, расположением изображений, видео и аудио.

Что такое HTML:

Дословно HTML означает Hypertext Markup Language (язык гипертекстовой разметки). Из расшифровки названия понятно, что инструмент применяется для управления отображением контента на интернет-странице, его структуризации.

Технология гипертекстовой разметки веб-страниц была предложена в 1989 году британским специалистом Тимом Бернерсом-Ли. Сначала язык применялся для обмена научной рабочей документацией между инженерами института CERN, сотрудником

которого был Бернерс-Ли. Немного позднее применение языка HTML было расширено настолько, что он, наряду с такими базовыми элементами, как HTTP и URL лег в основу Всемирной паутины.

Зачем нужен HTML:

Когда пользователь посещает сайт, браузер «подтягивает» файл HTML с данными о структуре и содержании веб-страницы. Функция HTML состоит в выстраивании внешней базы, фундамента, но сам запуск сайта в функционал не входит. HTML только указывает, где должны располагаться элементы, каков их базовый визуал, где брать стили для элементов и скрипты.

Возможности HTML:

HTML-документ можно составлять в любом редакторе, который есть в операционной системе: Notepad на MS Windows, TextEdit в Mac, Pico на Linux. Браузер для работы HTML–документа желателен, но необязателен. Он нужен для того, чтобы показать отформатированный документ.

Просматривать HTML-страницы можно и без выхода в интернет. Для этого нужно создать несколько HTML-файлов в одной папке, расположить в них гиперссылки и переходить по ним от одного документа к другому.

Что можно и нельзя сделать на HTML:

HTML представляет собой основу внутренней структуры сайта, его базовый каркас. Необходимо учитывать, что этот код является не языком программирования, как, например, Python или C#, а инструментом для разметки гипертекста. С его помощью браузер выстраивает интернет-страницу в виде, который понятен для людей, вырисовывает ее с помощью CSS и добавляя логику через JavaScript. HTML оптимален для начинающих программистов, он прост в освоении, а приобретенные навыки помогут уже в изучении языков программирования.

В HTML-файле можно задавать:

- гиперссылки;
- списки;
- формы;
- разметку страницы;
- таблицы;
- абзацы;
- картинки;
- видео;
- заголовки.

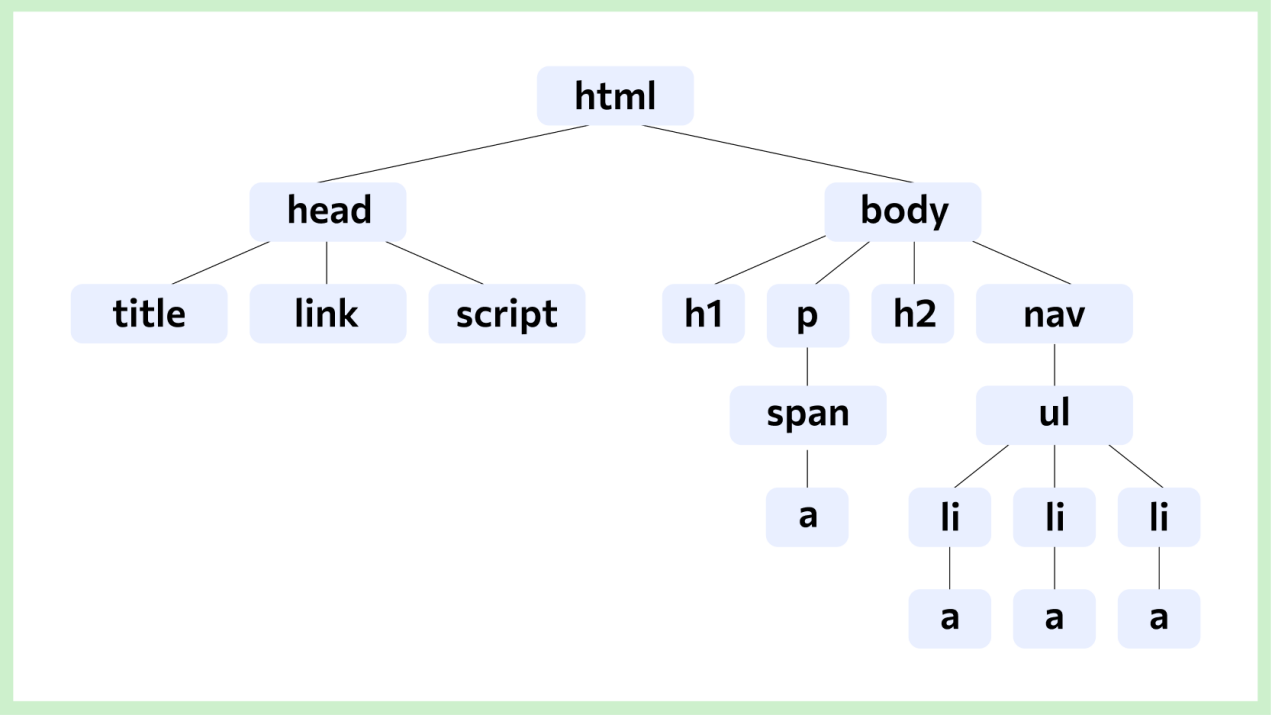
Создать базовый дизайн только с помощью HTML тоже можно. Например, установить цвет и шрифт текста на странице или фон для блоков. Использовать только код HTML для оформления веб-страниц не рекомендуется, дизайн будет примитивным и не современным. С CSS же творческий процесс ничем не ограничивается. Тем не менее, ряд функций в настоящий момент приходит в HTML из других, более серьезных инструментов. Например, Drag&Drop (перемещение элементов мышкой) ранее было исключительно в JavaScript, теперь это можно делать и на HTML.

Что такое теги HTML:

HTML-документ — это текстовый файл с расширением .html или .htm. В браузере он преобразуется в веб-страницу и состоит из набора тегов. Они как раз и помогают представлять текст на экране: благодаря им браузер понимает, что он читает не просто текст, а структурированную информацию, разбитую на блоки.

Тег выглядит как набор символов, заключенный в угловые скобки. Символы в скобках обозначают имя тега, которое описывает его функции. Вот несколько примеров:

- <h1> </h1> — заголовок;
-   — абзац;
-   — курсив.



Тег – это составной элемент, определяющий разметку структурных блоков. Он открывается, и этим начинает свое действие; и закрывается, обозначая завершение команды. Закрытые и открытые теги различаются только слешем перед именем тега. Эти теги создают оболочку, в которую помещается текст.

Именно незакрытые теги приводят к частым ошибкам и некорректным отображениям страницы. Для наглядности представим, что теги – это матрешки, из которых можно собрать набор. Складывая в большую матрешку все фигурки важно не забывать закрывать все половинки (ставить закрывающие теги), иначе игрушка не получится.

Внутри тега могут быть атрибуты – дополнительная информация, которую нужно скрыть из основного текста. Они ставятся только в открывающий тег, между ним и именем тега должен быть пробел, а после него идет знак равенства. Значение атрибута заключается в кавычки. С их помощью можно расширить возможности тегов и обратиться к ним, чтобы узнать подробную информацию.

Есть теги, которые нет необходимости закрывать. Пример: тег переноса строки
 — он одиночный и закрывать его не нужно. Раньше одиночные теги писались с закрывающим слэшем перед закрывающей скобкой. Например: <br />. В стандарте HTML5 использование закрывающего слеша в одиночных тегах необязательно. Примеры одиночных тегов: <br>, <hr>, <img>.

Помимо атрибутов в тег можно добавлять вложения, эти элементы могут менять стиль текста. Например, можно выделить какое-то слово **жирным** шрифтом.

Как выглядит код на HTML:

```
<!DOCTYPE html>
<html>
   <head>
     <title>Привет!</title>
   </head>
  <body>
     <img    src="https://static.tildacdn.com/tild6565-6361-4831-a664-366538616534/SF_MRG_logo_new.svg">
   </body>
</html>
```

<!DOCTYPE html> –предназначается для указания типа документа, так как браузер может интерпретировать разные версии HTML (например, EXtensible HyperText Markup Language, расширенный язык разметки гипертекста). По умолчанию его всегда включают в начало страницы.

<html> </html> – сообщает браузеру, что это за HTML-документ. Этот тег хранит в себе остальные теги.

<head> </head> – нужен для хранения других элементов, которые помогают браузеру в работе с данными. Внутри него есть метатеги, которые применяются, чтобы сохранять информацию для браузеров и поисковых систем.

<body> </body> – тело документа, в котором находятся все видимые пользователю элементы.

<title> </title> – заголовок веб-страницы. Именно его браузер загрузит как название, а при сохранении страницы в избранное он использует эту фразу как описание закладки.

<img> – помещает изображение в нужное место. Обычно к нему добавляют атрибут src, в котором содержится путь к этому изображению. Атрибуты width, height определяют ширину и высоту изображения в пикселях.

Основная разметка HTML-страницы – это заголовки, абзацы и списки. Они структурируют информацию на странице, все как в документе Word.

Заголовки:

В HTML бывает шесть уровней заголовков: <h1> – <h6>.

<h1>Привет!</h1>

<h2>Расскажешь</h2>

<h3>Какие бывают</h3>

<h4>Уровни заголовка</h4>

Заголовок типа <h1> используют обычно один раз, потому что он основной.

Абзац:

Как и на обычном письме, делит текст по смыслу.

Спасибо, всё понятно. Давай дальше

Списки:

Самые распространенные типы списков нумерованные и ненумерованные.

Ненумерованные или маркированные списки добавляются тегом . Такие списки применяются, когда нам не важна последовательность их элементов.

В нумерованном списке, где пункты расположены в определенном порядке, используется тег .

Отдельные элементы в любом типе списков заводятся тегом , который также нужно закрывать после каждого пункта.

Преимущества и недостатки HTML:

Преимущества:

- широкое распространение;
- совместимость с подавляющим числом браузеров;
- доступность;
- поддержка стандарта консорциумом Всемирной паутины (WWW Consortium);
- простая интеграция с базовыми языками программирования, такими как PHP.

Недостатки:

- не подходит для создания динамических страниц. Для этого может понадобиться JavaScript или PHP;
- некоторые браузеры медленно осваивают поддержку новых функций;
- иногда бывает сложно предугадать реакцию старых браузеров (Internet Explorer версии 8 и более ранней) на новые теги.

Является ли HTML языком программирования?

HTML не обрабатывает данные, а только их отображает. То есть с помощью него нельзя выполнить сложение или умножение, можно только показать текст, в котором будет

содержаться нужная формула с ответом. Он отвечает за разметку – ограниченный набор действий, который помогает браузеру отображать страницы.

Однако HTML обладает синтаксисом, семантикой и лексикой, поэтому он попадает в категорию декларативных языков программирования.

CSS:

CSS (от английского Cascading Style Sheets) — это «каскадные таблицы стилей». Являются формальным языком для задания дизайна документа на HTML или в других веб-форматах (XHTML, XML). Другими словами, этот язык описывает, как именно HTML-компоненты страницы должны отображаться на экране или других носителях, например, бумажных. Сегодня большая часть сайтов в глобальном интернете работает именно благодаря каскадным таблицам стилей. Также термин CSS используется для обозначения файла «стилей» сайта — такой файл не может использоваться отдельно от HTML-файла сайта. CSS-файл экономит время веб-программиста и верстальщика, так как позволяет оптимизировать управление дизайном для всех страниц сайта. Внешние таблицы стилей хранятся в CSS-файле, обычно он один для всего сайта.

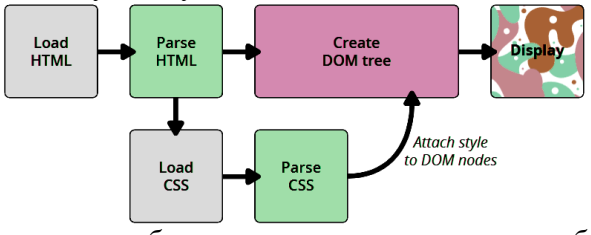
Простыми словами, CSS — это язык описания внешнего вида веб-страницы, который используется для настройки: цветов, типографики, местоположения компонентов страницы, стилей элементов и любого другого дизайна компонентов страницы.

Как работает CSS:

В HTML-документе находятся данные только о структуре веб-страницы. Чтобы правильно вывести её на экран, с учётом дизайна сайта и особенности экрана пользовательского устройства, браузер объединяет HTML-документ с файлом стилей, который есть у любого сайта. Такое объединение происходит поэтапно. Вот основные шаги, как работает CSS:

1. Пользователь открывает сайт, например, переходя на него со страницы результатов поиска.
2. Браузер начинает загрузку HTML-документа.
3. Файл преобразуется в DOM (объектная модель документа в оперативной памяти).
4. Браузер анализирует все компоненты, на которые в HTML-документе есть URL, и которые связаны с этим документом. К таким ресурсам и компонентам как раз и относятся стили (а также, например, любые медиафайлы: картинки, GIF, видеофайлы). Внимание: если в HTML-документе ссылки на стили нет, то браузер не сможет получить к ним доступ. JS-код обрабатывается далее, но не на этом этапе.
5. Браузер начинает проверять файл стилей. В частности, он пытается отсортировать правила, содержащиеся в нём (по их типу селектора). Каждое правило определяется в свою категорию (например, ID, элемент, класс). Далее, на этом промежуточном шаге, также происходит связывание обнаруженных селекторов с правилами. Правила применяются к определенным DOM-узлам, а затем к каждому из них привязывается определённый стиль. Всё, теперь браузер знает, как именно нужно отрисовывать страницу.
6. Происходит отрисовка страницы уже с настроенным дизайном её элементов.

Если вышеуказанная последовательность работы CSS показалась для вас слишком сложный, посмотрите на эту схему:



Последовательность работы со стилями элементов в браузере.

Вообще существует не один, а несколько методов сформировать правила «стилей». Это не только задание набора свойств с фиксированными значениями, но и метод при помощи селекторов, например.

Основы:

Синтаксис CSS:

Начнём с подключения стилей к веб-странице. Файл стилей может публиковаться разными способами, внутренними и внешними. Самый частый сценарий подключения CSS — самостоятельный файл, который затем подключается к веб-странице через тег link:

```
<!DOCTYPE html>
<html>
   <head>
      .....
     <link rel="stylesheet" type="text/css" href="style.css">
   </head>
   <body>
      .....
   </body>
</html>
```

Другой способ подключения стилей к сайту — самостоятельный, без файла-родителя. Для этого можно использовать директиву import в контейнере style:

```
<!DOCTYPE html>
<html>
   <head>
      .....
      <style media="all">
         @import url(style.css);
      </style>
   </head>
</html>
```

При этом CSS может быть указан непосредственно внутри документа: (опять же, обратите внимание на тег style внутри head)

```
<!DOCTYPE html>
<html>
   <head>
      .....
      <style>
         body {
            color: red;
         }
      </style>
   </head>
   <body>
      .....
   </body>
</html>
```

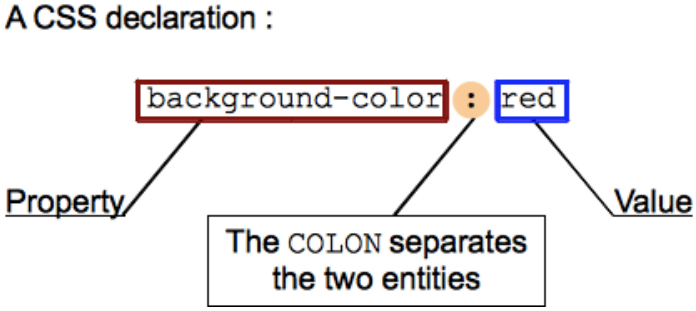
Наконец, CSS может быть указан в атрибуте style:

```
<!DOCTYPE>
<html>
   <head>
      .....
   </head>
   <body>
      <p style="font-size: 20px; color: green; font-family: arial,
helvetica, sans-serif">
         .....
      </p>
   </body>
</html>
```

Теперь давайте посмотрим на важные особенности синтаксиса CSS. Напомним, главная цель CSS — передать браузеру инструкции, согласно которым он будет отрисовывать страницу. Согласно этой идее главными будут являться два элемента: property и value.

1. Первый элемент (property) — это непосредственное свойство, которое и будет изменено.
2. Второй элемент (value) — это значение, которым движок браузера будет обрабатывать свойство. Вот от этих двух блоков и строится дальнейший синтаксис «стилей».

Объявления. Два вышеуказанных блока (property and value) используются в качестве *объявлений*.



Так работает объявление или declaration в CSS.

Поиск совпадений между такими объявлениями и элементами — основная задача движка «стилей». Посмотрите на этот пример синтаксиса CSS:

```
selectorlist { property: value; [more property:value; pairs] }
```

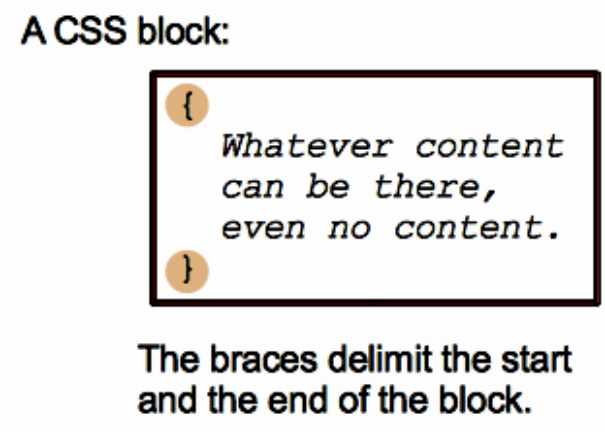
Как видим для списка селекторов указываются: свойство, затем его значение, затем — добавляются аналогичные пары. Здесь всё просто. Но большое разнообразие свойств и огромное количество допустимых значений, часто приводят к тому, что новички допускают ошибки в синтаксисе (и не только). А если выбрано некорректное значение для какого-либо элемента, то браузер не будет выполнять такое объявление.

Ещё один простой пример синтаксиса CSS:

```
strong { color: red; }
div.menu-bar li:hover > ul { display: block; }
```

Важно: данные внутри пары свойство / значение всегда зависимы от регистра. Пробелы, при этом, игнорируется (в любых местах). Сама пара всегда отделяется при помощи знака «:» (двоеточие).

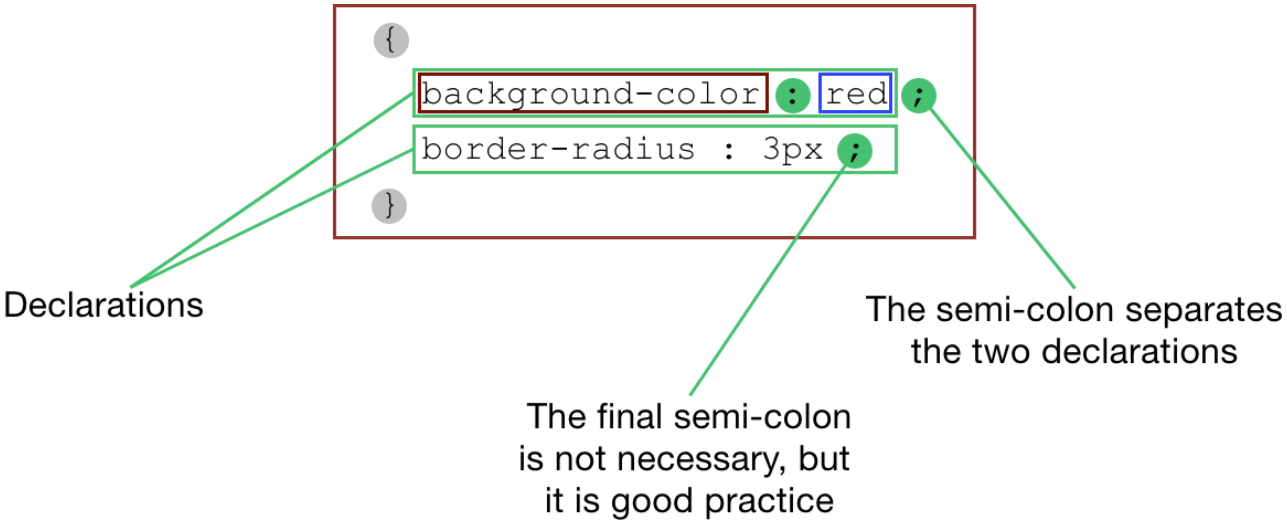
Блоки. Традиционно объявления в CSS объединяются в *блоки*. Их синтаксис подразумевает открывающую и закрывающую фигурную скобку. Схематично вышесказанное можно представить вот так:



Между фигурными скобками располагается содержимое.

Блок объявлений. Подразумевает отделение самих объявлений внутри при помощи знака ;

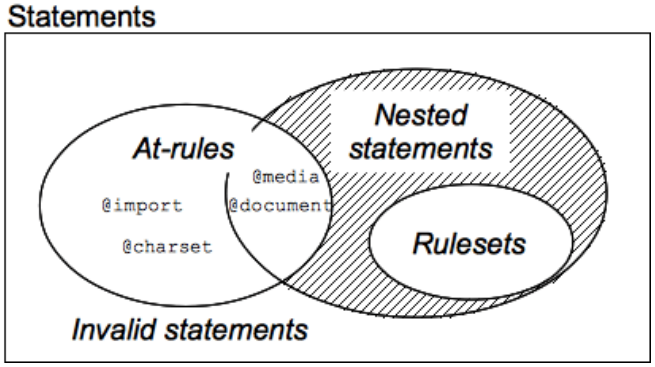
Вот как работают объявления, блоки и блоки объявлений друг с другом:



Первым указываются объявления. Точка с запятой — используется для разделения двух объявлений.

Операторы CSCS (они же statements):

Наборы правил — своего рода кирпичики для постройки финального файла стиля для сайта. Обычно — это длинный-длинный список из разных параметров.



Пример пересечения действия операторов at-rules и rulesets.

Но не только данные о внешнем виде элементов страницы передают в «стилях». Также есть технические сведения, которые должны передаваться движку браузера (например, данные о счетчиках веб-аналитики, наборах, настройках типографики и другое). Для таких данных задействуются иные, специфические виды утверждений. Это, в первую очередь, at-rules и rulesets (наборы правил).

At-rules. Их характерный маркер — наличие символа @ в начале. Например, вот так:

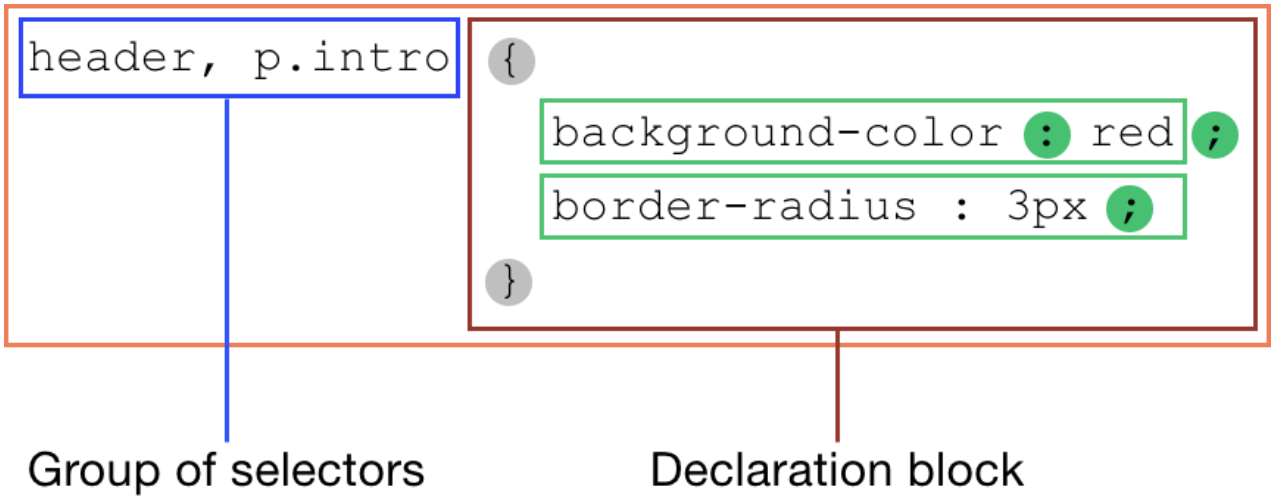
```
/* Наша структура */
@ID (ПРАВИЛО);
/* Пример: приказывает браузеру задействовать UTF-8 кодировку*/
@charset "utf-8";
```

Примеры:

- @charset — установка кодировки. Определяет кодировку символов, используемую таблицей «стилей».
- @import — нужно задействовать внешний файл CSS.
- @namespace — содержимое документа должно интегрироваться как для XML (только пространство имён).

Существуют и вложенные at-rules. Это множество CSS-операторов, которое допустимо задействовать в качестве разных правил в какой-либо группе или как самостоятельный оператор-CSS. Примеры вложенных операторов: @media, @page, @viewport, @supports.

Rulesets (наборы правил). Применяемость CSS файла по отдельности к каждому элементу на странице — это хорошо, но неудобно. Именно поэтому в «стиле» подразумевают применение любых объявлений к любым частям веб-страницы. В «стилях» вы можете объединить наборы правил с конкретными блоками объявлений.



Чтобы соединить наборы правил с блоками объявлений, каждому блоку должен предшествовать селектор.

Парные селекторы объявления — это и есть набор правил CSS. Но здесь есть некоторые сложности, например: часто определённому элементу в документе релевантны сразу несколько селекторов. Для определения приоритета иерархичности используется каскадный алгоритм. Именно он уточнит иерархию какого-либо свойства, если оно упоминается с различными значениями.

О селекторах, комбинаторах и псевдоэлементах CSS:

Селектор устанавливает точное правило, которое будет использоваться для конкретного элемента веб-страницы.

В стилях CSS распространено большое количество контейнеров: по атрибуту, универсальный, по ID, по типу элемента, по классу. Также для работы с селекторами

предусмотрены комбинаторы и псевдоэлементы (используются для выбора того, что отсутствует в HTML-коде). Что касается комбинаторов, то они упрощают выбор элементов. Например, комбинатор adjacent sibling — позволяет выбирать последующие соседние элементы, но только если у них имеется общий родитель.

Комбинатор comma (в качестве него используется знак запятой). Он позволяет сгруппировать и выделить все идентичные узлы следующего соседнего элемента. Также есть отдельные комбинаторы для всех соседних элементов, для потомков, дочерних элементов.

Важно: при помощи селекторов вы не сможете выбрать родительский элемент с контейнером внутри.

Разбираем пример CSS верстки:

Чтобы лучше усвоить всю информацию выше — предлагаем изучить реальный пример файла стилей для сайта.

```
p {
   font-family: Liberation Sans, helvetica, sans-serif;
}
h2 {
   font-size: 22pt;
   color: green;
   background: white;
}
.note {
   color: green;
   background-color: white;
   font-weight: bold;
}
p#paragraph1 {
   padding-left: 12px;
}
a:hover {
   text-decoration: none;
}
#news p {
   color: green;
}
[type="button"] {
   background-color: green;
}
```

Давайте посмотрим, что есть в этом коде и как его расшифровать.

Сразу в глаза бросается наличие семи CSS-правил с различающимися селекторами. Рассмотрим их подряд сверху вниз, также, как они расписаны в этом коде.

1. Правило для абзаца обозначено привычным элементом p. Шрифт по умолчанию — Liberation Sans, но если он не может быть загружен, то будут использоваться другие шрифты.

```
p {
   font-family: Liberation Sans, helvetica, sans-serif;
}
```

Если Liberation Sans будет недоступен, его заменит Helvetica.

2. Правило для заголовка второго уровня обозначено элементом h2. Начертание такого заголовка должно иметь зеленый цвет, а бэкграунд — белый цвет. Обратите внимание, что указан весьма крупный шрифт (22 pt).

```
h2 {
   font-size: 22pt;
   color: green;
   background: white;
}
```

Это правило для всех H2-заголовков.

3. Правило для одного или нескольких элементов, имеющих класс note. Устанавливает зеленый цвет в качестве основного и белый в качестве фонового цвета. Кроме того, это правило задаёт элементам полужирное начертание.

```
}
.note {
   color: green;
   background-color: white;
   font-weight: bold;
}
```

Настраиваем внешний вид элементам с классом note.

4. Правило для конкретного элемента p (с идентификатором paragraph1). Устанавливает для указанного элемента отступ в 12 px.

```
}
p#paragraph1 {
   padding-left: 12px;
}
```

Настраиваем отступ для параграфа.

5. Правило убирает подчеркивание по умолчанию во всех ссылках при наведении курсора на такой элемент.

```
}
a:hover {
   text-decoration: none;
}
#news p {
```

Убираем подчеркивание при наведении на ссылки.

6. Правило для конкретных элементов p, расположенных внутри другого компонента с идентификатором news.

```
}
#news p {
    color: green;
}
```

Теперь этот элемент будет иметь зеленый цвет.

7. Правило только для элементов, имеющих атрибут type со значением button. Устанавливает зеленый цвет фона.

```
}
[type="button"] {
    background-color: green;
}
```

Поменяли цвет фона у кнопок на зеленый.

Методологии CSS:

Методологии CSS регулируют способы работы и написания кода. Они также устанавливают то, как именно будет выглядеть итоговый код и по каким правилам он должен писаться.

В разное время существовали несколько популярных методологий. Часть из них — продолжает жить, другая часть — была забыта по разным причинам. Упоминать все существующие когда-либо методологии — не имеет смысла. Ведь в конце 2022 — начале 2023 разработчики активно используют только пять из них:

BEM. Вся суть этой методологии сразу становится понятна, если знать, как расшифровывается это название — модификатор блочных элементов. Посмотрите на этот код:

```
<form class="loginform loginform--errors">
    <label class="loginform__username loginform__username--error">
        Username <input type="text" name="username" />
    </label>
    <label class="loginform__password">
        Password <input type="password" name="password" />
    </label>
    <button class="loginform__btn loginform__btn--inactive">
        Sign in
    </button>
</form>
```

При помощи BEM мы формируем такие компоненты, которые можно задействовать повторно. Вернёмся к примеру выше. Обратите внимание — класс *loginform* включает в себя три компонента:

- loginform__username (используется для ввода имени пользователя).
- loginform__password (используется для ввода пароля пользователя).
- loginform__btn (используется для того, чтобы пользователь мог повторно отправить форму).

Systematic, она же систематическая. В этой методологии можно обнаружить большую часть принципов, заложенных в популярных SUIT CSS, OOCSS, SMACSS и конечно BEM. Systematic CSS — довольно интуитивная методология, которая является хорошей альтернативой искусственно усложненным системам. Вот как может выглядеть код CSS, согласно этой методологии, для вывода панели навигации и поисковой формы.

```
<!-- navigation bar -->
<div class="NavBar">
   <ul>
      <li><a href="./">Home</a></li>
      <li><a href="about.html">About</a></li>
      <li><a href="learn/">Learn</a></li>
      <li><a href="extend/">Extend</a></li>
      <li><a href="share/">Share</a></li>
   </ul>
</div>

<!-- search form -->
<div class="SearchBox">
   <form action="search.html" method="get">
       <label for="input-search">Search</label>

       <input name="q" type="search" id="input-search" />
       <button type="submit">Search</button>
    </form>
</div>
```

Контент — в виде виджетов и открытых HTML-элементов — затем помещается в макет.
Наконец, добавляются классы-модификаторы, чтобы изменить представление вещей по умолчанию.

Пример кода по методологии Systematic.

Объектно-ориентированная, она же OOCSS. Хорошо продуманная методология, которая отличается гибкостью и возможностью заменять некоторые компоненты. Нужно просто привыкнуть к ней. Набор кода станет более логичным и прозрачным. Модульность — ещё одна черта OOCSS. Пример:

```
.button {
    box-sizing: border-box;
    height: 60px;
    width: 90%;
}

.grey-btn {
    background: #EEE;
    border: 2px solid #DDD;
    box-shadow: rgba(0, 0, 0, 0.5) 2px 2px 4px;
    color: #666;
```

Как видим, мы установили стиль кнопки при помощи двух классов (button используем для показа структуры, а gret-btn — для настройки дизайна элемента).

HTML-файл, с учетом вышесказанного, будет выглядеть следующим образом:

```
<button class="button grey-btn">
    Нажми здесь!
</button>
```

SMA CSS. Неплохая методология для CSS. Чтобы не объяснять словами, сразу проведём пример кода, который хорошо иллюстрирует особенности этой методологии:

```
<section class="our-footer">
    <form class="search is-submitted">
        <input type="search" />
        <input type="button" value="Search">
    </form>
</section>
```

Atomizer-подход (назван в честь одноименной CSS-библиотеки). Вот несколько маркеров этой методологии: имена базируются на внешнем поведении функции, а одноцелевые классы — становятся основными строительными блоками. Но вместо подробного описания, лучше посмотрите на пример ниже:

```
<div class="Bgc(#0280ae.5) C(#fff) P(20px)">
    Primer
</div>
```

Мы определили цвет, задав значение в шестнадцатеричной системе. Примечательно, как реализован opacity-канал — он сделан через применение одноименного параметра к hex-цвету.

Резюмируя:

Ключевые различия между HTML и CSS:

- HTML является основным языком разметки, который описывает содержание и структуру веб-страниц. С другой стороны, CSS является расширением HTML, которое изменяет дизайн и отображение веб-страниц.
- HTML-файл может содержать код CSS, в то время как таблицы стилей CSS никогда не могут содержать HTML-код.
- HTML состоит из **тегов,** окружающих контент. В то время как CSS состоит из **селекторов, за** которыми следует **блок объявления**.

Преимущества HTML:

- Простой в использовании и имеющий свободный синтаксис (хотя, будучи слишком гибким, не будет соблюдать стандарты).
- Широко используется, устанавливается практически на каждом веб-сайте и поддерживается каждым браузером.
- Аналог синтаксиса XML, который все чаще используется для хранения данных.
- Это бесплатно, так как вам не нужно покупать программное обеспечение.
- Легко учиться и писать код даже новичкам.

Преимущества CSS:

- CSS экономит ваше время, написав CSS один раз и повторно используя один и тот же лист на нескольких страницах.
- Страницы требуют меньше времени для загрузки из-за меньшего количества кода.
- Простота в обслуживании, глобальные изменения просты в использовании.
- CSS имеет лучшие стили для HTML и гораздо более широкий диапазон атрибутов.
- Обеспечение совместимости нескольких устройств.
- Теперь атрибуты HTML осуждаются, и рекомендуется использовать CSS на всех страницах HTML, чтобы сделать их совместимыми с будущими браузерами.
- Поддержка автономного просмотра с помощью автономного кэша.
- Скрипт обеспечивает постоянную независимость от платформы и может поддерживать новейшие браузеры.

Недостатки HTML:

- Поскольку это статический язык, он не может генерировать динамический вывод.
- Предлагает ограниченные функции безопасности.

Недостатки CSS:

Фрагментация - CSS отображает разные размеры в каждом браузере. Программисты должны рассмотреть и протестировать весь код в нескольких браузерах, прежде чем запускать любой веб-сайт или мобильное приложение, чтобы не возникало проблем с совместимостью.

HTML и CSS — два основных языка, обычно используемых для веб-разработки. **Разница** между **HTML и CSS** заключается в том, что **HTML** — это язык разметки, который используется для создания структуры веб-страницы, а **CSS** — это язык стилей, который используется для того, чтобы сделать веб-страницы более презентабельными.

SQL:

SQL (**Structured Query Language**, или язык структурированных запросов) — это декларативный язык программирования (язык запросов), который используют для создания, обработки и хранения данных в реляционных БД.

На чистом SQL нельзя написать программу — он предназначен только для взаимодействия с базами данных: получения, добавления, изменения и удаления информации в них, управления доступом и так далее.

Реляционная база данных – это **набор данных с предопределенными связями между ними.** Эти данные организованны в виде набора таблиц, состоящих из столбцов и строк. В таблицах хранится информация об объектах, представленных в базе данных. В каждом столбце таблицы хранится определенный тип данных, в каждой ячейке – значение атрибута. Каждая стока таблицы представляет собой набор связанных значений, относящихся к одному объекту или сущности.

SELECT и FROM:

SELECT в запросе определяет, какие столбцы данных отобразить в результатах. Кроме того, в SQL есть возможности отображать данные не из столбца таблицы. В примере ниже показаны 3 столбца, взятые из таблицы студентов Student (через SELECT и FROM) и один вычисляемый столбец. В базе данных хранятся ID (studentID), имя (FirstName) и фамилия (LastName) студента. Мы можем объединить столбцы с именем и фамилией и создать вычисляемое поле с полным именем (FullName).

SELECT studentID, FirstName, LastName, FirstName + ' ' + LastName AS FullName FROM student;

Вывод:

```
+-----------+-------------------+------------+------------------------+
| studentID | FirstName         | LastName   | FullName               |
+-----------+-------------------+------------+------------------------+
|         1 | Monique           | Davis      | Monique Davis          |
|         2 | Teri              | Gutierrez  | Teri Gutierrez         |
|         3 | Spencer           | Pautier    | Spencer Pautier        |
|         4 | Louis             | Ramsey     | Louis Ramsey           |
|         5 | Alvin             | Greene     | Alvin Greene           |
|         6 | Sophie            | Freeman    | Sophie Freeman         |
|         7 | Edgar Frank "Ted" | Codd       | Edgar Frank "Ted" Codd |
|         8 | Donald D.         | Chamberlin | Donald D. Chamberlin   |
|         9 | Raymond F.        | Boyce      | Raymond F. Boyce       |
+-----------+-------------------+------------+------------------------+
9 rows in set (0.00 sec)
```

CREATE TABLE:

Название CREATE TABLE говорит само за себя – оператор создает таблицу. Вы можете задать название таблицы и настроить, какие столбцы будут присутствовать в таблице.

CREATE TABLE table_name (column_1 datatype, column_2 datatype, column_3 datatype);

ALTER TABLE:

ALTER TABLE изменяет структуру таблицы. Вот так можно добавить столбец в базу данных:

ALTER TABLE table_name ADD column_name datatype;

CHECK:

CHECK ограничивает диапазон значений, которые можно добавить в столбец.

Когда вы настраиваете ограничение CHECK для отдельного столбца, оператор проверяет, что в этом столбце присутствуют строго определенные значения. Если же CHECK настраивается для таблицы, то он может ограничивать значения в отдельных столбцах на основании значений из других столбцов этой строки.

В следующем примере при создании таблицы Persons используется ограничение CHECK для столбца «Возраст» (Age). Таким образом проверяется, что в таблицу не попадают лица младше 18 лет.

CREATE TABLE Persons (ID int NOT NULL, LastName varchar (255) NOT NULL, FirstName varchar (255), Age int, CHECK (Age>=18));

Следующий синтаксис используется для присвоения названия оператору CHECK и настройки CHECK для нескольких столбцов:

CREATE TABLE Persons (ID int NOT NULL, LastName varchar (255) NOT NULL, FirstName varchar (255), Age int, City varchar (255), CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes'));

WHERE (AND, OR, IN, BETWEEN и LIKE):

Оператор WHERE используется для ограничения количества возвращаемых строк.

Сначала, в качестве примере, мы покажем оператор SELECT и его результат без оператора WHERE. Затем добавим оператор WHERE, в котором используются сразу 5 из вышеуказанных квалификаторов.

SELECT studentID, FullName, sat_score, rcd_updated FROM student;

```
+-----------+-------------------------+-----------+---------------------+
| studentID | FullName                | sat_score | rcd_updated         |
+-----------+-------------------------+-----------+---------------------+
|         1 | Monique Davis           |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez          |       800 | 2017-08-16 15:34:50 |
|         3 | Spencer Pautier         |      1000 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey            |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene            |      1200 | 2017-08-16 15:34:50 |
|         6 | Sophie Freeman          |      1200 | 2017-08-16 15:34:50 |
|         7 | Edgar Frank "Ted" Codd  |      2400 | 2017-08-16 15:35:33 |
|         8 | Donald D. Chamberlin    |      2400 | 2017-08-16 15:35:33 |
|         9 | Raymond F. Boyce        |      2400 | 2017-08-16 15:35:33 |
+-----------+-------------------------+-----------+---------------------+
9 rows in set (0.00 sec)
```

Теперь повторим запрос SELECT, но ограничим возвращаемые строки оператором WHERE.

```
+-----------+----------------------+-----------+---------------------+
| studentID | FullName             | sat_score | rcd_updated         |
+-----------+----------------------+-----------+---------------------+
|         1 | Monique Davis        |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez       |       800 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey         |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene         |      1200 | 2017-08-16 15:34:50 |
|         8 | Donald D. Chamberlin |      2400 | 2017-08-16 15:35:33 |
+-----------+----------------------+-----------+---------------------+
5 rows in set (0.00 sec)
```

UPDATE:

Для обновления записи в таблице используется оператор UPDATE.

Условием WHERE можно уточнить, какие именно записи вы бы хотели обновить. Вы можете обновлять по одному или нескольким столбцам сразу. Синтаксис выглядит так:

UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;

В примере ниже обновляется название записи (поле Name) с Id 4:

UPDATE Person SET Name = “Elton John” WHERE Id = 4;

Помимо этого, можно обновлять столбцы в таблице значениями из других таблиц. Чтобы получить данные из нескольких таблиц, воспользуйтесь оператором JOIN. Синтаксис выглядит так:

UPDATE table_name1 SET table_name1.column1 = table_name2.columnA table_name1.column2 = table_name2.columnB FROM table_name1 JOIN table_name2 ON table_name1.ForeignKey = table_name2.Key

В примере ниже мы обновляем поле «Менеджер» (Manager) для всех записей:

UPDATE Person SET Person.Manager = Department.Manager FROM Person JOIN Department ON Person.DepartmentID = Department.ID

GROUP BY:

GROUP BY позволяет объединять строки и агрегировать данные.

Вот так выглядит синтаксис GROUP BY:

SELECT column_name, COUNT(*) FROM table_name GROUP BY column_name;

HAVING:

HAVING позволяет сортировать данные, которые собираются через GROUP BY. Таким образом, пользователю показывается лишь ограниченный набор записей.

Вот так выглядит синтаксис HAVING:

SELECT column_name, COUNT(*) FROM table_name GROUP BY column_name HAVING COUNT(*) > value;

AVG():

AVG, или среднее, вычисляет среднее значение числового столбца из набора строк, которые возвращает оператор SQL.

Вот так выглядит синтаксис:

SELECT groupingField, AVG(num_field) FROM table1 GROUP BY groupingField

А вот пример этого оператора для таблицы Student:

SELECT studentID, FullName, AVG(sat_score) FROM student GROUP BY studentID, FullName;

AS:

AS позволяет переименовать столбец или таблицу с помощью псевдонима.

SELECT user_only_num1 AS AgeOfServer, (user_only_num1 - warranty_period) AS NonWarrantyPeriod FROM server_table

И вот так будет выглядеть результат.

STUDENT studentID, FullName, sat_score, recordUpdated FROM student WHERE (studentID BETWEEN 1 AND 5 OR studentID = 8) AND sat_score NOT IN (1000, 1400);

AgeOfServer	NonWarrantyPeriod
36	24
24	12
61	49
12	0
6	-6
0	-12
36	24
36	24
24	12

Кроме того, через оператор AS вы можете задать название таблицы – так будет проще обращаться к ней в JOIN.

SELECT ord.product, ord.ord_number, ord.price, cust.cust_name, cust.cust_number FROM customer_table AS cust JOIN order_table AS ord ON cust.cust_number = ord.cust_number

Результат выглядит так.

product	ord_number	price	cust_name	cust_number
RAM	12345	124	John Smith	20
CPU	12346	212	Mia X	22
USB	12347	49	Elise Beth	21
Cable	12348	0	Paul Fort	19
Mouse	12349	66	Nats Back	15
Laptop	12350	612	Mel S	36
Keyboard	12351	24	George Z	95
Keyboard	12352	24	Ally B	55
Air	12353	12	Maria Trust	11

ORDER BY:

ORDER BY позволяет сортировать результирующий набор данных по одному или нескольким элементам в разделе SELECT. Ниже дан пример сортировки студентов по имени (FullName) в порядке убывания. Изначально используется стандартная сортировка по возрастанию (ASC), поэтому для сортировки в обратном порядке мы применяем DESC.

SELECT studentID, FullName, sat_score FROM student ORDER BY FullName DESC;

COUNT:

COUNT вычисляет количество строк и возвращает результирующее значение в столбце.

Ниже приводятся возможные сценарии использования COUNT:

- Подсчет всех строк в таблице (не требуется Group by)
- Подсчет общего числа подмножеств данных (в операторе обязательно прописывается Group By)

Этот оператор SQL выводит количество всех строк. Кроме того, что вы можете настроить название результирующего столбца COUNT с помощью AS.

```
SELECT count(*) AS studentCount FROM student;
```

DELETE:

DELETE используется для удаления записи из таблицы.

Будьте внимательны! Вы можете удалить несколько записей в таблице, либо сразу все. С помощью условия WHERE вы указываете, какие записи необходимо удалить. Синтаксис выглядит так:

```
DELETE FROM table_name WHERE condition;
```

Вот так выглядит удаление из таблицы Person записи с Id 3:

```
DELETE FROM Person WHERE Id = 3;
```

INNER JOIN:

JOIN, или внутреннее соединение, выбирает записи, соответствующие значениям в двух таблицах.

```
SELECT * FROM A x JOIN B y ON y.aId = x.Id
```

LEFT JOIN:

LEFT JOIN возвращает все строки из левой таблицы и соответствующие им строки из правой таблицы. Строки из левой таблицы возвращаются даже при пустых значениях в правой таблице. Если для строк из левой таблицы нет соответствия в правой, то в значениях последней будет стоять null.

```
SELECT * FROM A x LEFT JOIN B y ON y.aId = x.Id
```

RIGHT JOIN:

RIGHT JOIN возвращает все строки из правой таблицы и соответствующие им строки из левой. В отличие от левого соединения, здесь возвращаются все строки из правой таблицы, даже если им ничего не соответствует в левой. В таком случае, в значениях столбцов из левой таблицы будет стоять null.

```
SELECT * FROM A x RIGHT JOIN B y ON y.aId = x.Id
```

FULL OUTER JOIN:

FULL OUTER JOIN возвращает все строки, соответствующие условиям в любой из таблиц. Если в левой таблице есть строки, которым ничего не соответствует в правой, то они все равно отобразятся в результирующих значениях. То же самое распространяется и на строки из правой таблицы без соответствующих значений в левой.

```
SELECT Customers.CustomerName, Orders.OrderID FROM Customers FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID ORDER BY Customers.CustomerName
```

INSERT:

INSERT используется для добавления данных в таблицу.

```
INSERT INTO table_name (column_1, column_2, column_3) VALUES (value_1, 'value_2', value_3);
```

LIKE:

LIKE используется в связке с WHERE или HAVING (в составе оператора GROUP BY) и ограничивает выбранные строки по элементам, если в столбце содержится определенный шаблон символов.

Этот SQL запрос выбирает студентов, чье значение в FullName начинается с «Monique» или заканчивается с «Greene».

SELECT studentID, FullName, sat_score, rcd_updated FROM student WHERE FullName LIKE 'Monique%' OR FullName LIKE '%Greene';

```
+-----------+---------------+-----------+---------------------+
| studentID | FullName      | sat_score | rcd_updated         |
+-----------+---------------+-----------+---------------------+
|         1 | Monique Davis |       400 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene  |      1200 | 2017-08-16 15:34:50 |
+-----------+---------------+-----------+---------------------+
2 rows in set (0.00 sec)
```

Перед LIKE вы можете добавить NOT, и тогда строки, соответствующие условию, будут исключаться, а не добавляться. Этот SQL исключает записи, у которых в столбце FULL NAME содержится «cer Pau» и «Ted».

SELECT studentID, FullName, sat_score, rcd_updated FROM student WHERE FullName NOT LIKE '%cer Pau%' AND FullName NOT LIKE '%"Ted"%';

```
+-----------+---------------------+-----------+---------------------+
| studentID | FullName            | sat_score | rcd_updated         |
+-----------+---------------------+-----------+---------------------+
|         1 | Monique Davis       |       400 | 2017-08-16 15:34:50 |
|         2 | Teri Gutierrez      |       800 | 2017-08-16 15:34:50 |
|         4 | Louis Ramsey        |      1200 | 2017-08-16 15:34:50 |
|         5 | Alvin Greene        |      1200 | 2017-08-16 15:34:50 |
|         6 | Sophie Freeman      |      1200 | 2017-08-16 15:34:50 |
|         8 | Donald D. Chamberlin|      2400 | 2017-08-16 15:35:33 |
|         9 | Raymond F. Boyce    |      2400 | 2017-08-16 15:35:33 |
+-----------+---------------------+-----------+---------------------+
7 rows in set (0.00 sec)
```