***XML***

Язык **XML** предназначен для хранения и передачи данных. **HTML** же предназначен для отображения данных.

Что такое XML?

* **XML** - аббревиатура от англ. *eXtensible Markup Language* (пер. расширяемый язык разметки).
* **XML** – язык разметки, который напоминает HTML.
* **XML** предназначен для передачи данных, а не для их отображения.
* **Теги XML** не предопределены. Вы должны сами определять нужные теги.
* **XML** описан таким образом, чтобы быть самоопределяемым.

Разница между XML и HTML

XML не является заменой HTML. Они предназначены для решения разных задач: XML решает задачу хранения и транспортировки данных, фокусируясь на том, что такое эти самые данные, HTML же решает задачу отображения данных, фокусируясь на том, как эти данные выглядят. Таким образом, HTML заботится об отображении информации, а XML о транспортировке информации.

И при всем при этом этот документ XML не делает ничего. Это просто информация, обернутая в теги. Кто-то должен написать программу, которая будет отсылать, получать и отображать эти данные.

***Нативные vs Гибридные приложения***

**Веб-приложения**

Наиболее распространённый тип, который с лёгкостью позволяет отображать сайты на различных устройствах. Это не приложения в чистом виде, а адаптированные под мобильные устройства интерфейсы сайтов с активной функциональностью. Веб-приложения отличаются кросс-платформенностью, их не нужно устанавливать, и они достаточно просты в использовании. К другим преимуществам этого типа ПО относится достаточно простая и оперативная разработка. Но есть у веб-приложений и недостатки: необходимость подключаться к интернету для работы, ограниченная безопасность, невысокая производительность.

##### **Нативные приложения**

Такие приложения разработаны только под определённую платформу и по максимуму используют возможность той или иной операционной системы. Нативные приложения могут задействовать и иные ресурсы гаджета: камера, навигатор, список контактов и прочее. Они характеризуются широкой функциональностью и высокой скоростью работы.

Но есть у подобных приложений и недостатки: низкий по сравнению с мобильными веб-приложениями охват платформ, высокая стоимость разработки и необходимость регулярно выпускать обновления.

**Гибридные приложения**

Комбинация двух вышеупомянутых типов. В отличие от нативных, гибридные приложения разрабатываются для нескольких платформ одновременно и пишутся на универсальном языке. Такой продукт можно размещать в сторах, а для его обновления не нужно каждый раз выпускать новую версию. Достаточно лишь добавить все изменения на сервер. Слабое место подобных приложений ― визуальный стиль. Ведь разработка приложения для конкретной платформы ведётся на основе единого гайдлайна. А интерфейс гибридного приложения теряет узнаваемые черты операционной системы.

Конечно, разработка таких приложений обходится дешевле, но и потенциально уязвимых мест больше.

***Тестирование мобильных приложений***

8. Тестирование pop-up, алертов  
9. Тестирование WebView  
10. Скролл/свайп элементов  
11. Тестирование PUSH уведомлений  
12. Сворачивание/разворачивание приложения  
13. Разные типы подключений (сотовая связь/Wi-Fi)  
14. Ориентация экрана (альбомная/портретная)  
15. Темная/светлая темы  
16. Реклама в приложении  
17. Шаринг контента в соц. сети  
18. Работа приложения в фоне  
20. Политики конфиденциальности и прочие ссылки на документы

### Тестирование совместимости

1. Корректное отображение гео  
3. Различные способы оплаты (Google Pay, Apple Pay)  
4. Тестирование датчиков (освещенности, температуры устройства, гироскоп и т.д.)  
5. Тестирование прерываний (входящий звонок/смс/push/будильник/режим «Не беспокоить» и т.д.)  
6. Подключение внешних устройств (карта памяти/наушники и т.д.)

### Тестирование безопасности

1. Тестирование разрешений (доступ к камере/микрофону/галерее/и т.д.)  
2. Данные пользователя (пароли) не передаются в открытом виде  
3. В полях, с вводом пароля и подтверждением пароля, данные скрываются астерисками

### Тестирование локализации и глобализации

#### Что проверяем?

1. Все элементы в приложении переведены на соответствующий язык  
2. Тексты зашиты внутри приложения и пользователь в настройках приложения может выставить необходимый язык  
3. Тексты зависят от языка в системных настройках  
4. Тексты приходят с сервера  
5. Корректное отображение форматов дат (ГОД — МЕСЯЦ — ДЕНЬ или ДЕНЬ — МЕСЯЦ — ГОД.)  
6. Корректное отображение времени в зависимости от часового пояса

9. Анимация между переходами  
10. Корректный возврат на предыдущий экран  
11. Поддерживаются основные жесты при работе с сенсорными экранами (swipe back и т.д.)

### Стрессовое тестирование

Стрессовое тестирование направлено на определение эффективности производительности приложения в условиях повышенной нагрузки. Стресс-тест в этом контексте ориентирован только на мобильные устройства.

#### Что проверяем?

1. Высокая загрузка центрального процессора  
2. Нехватка памяти  
3. Загрузка батареи  
4. Отказы  
5. Низкая пропускная способность сети  
6. Большое количество взаимодействий пользователя с приложением (для этого может понадобиться имитация реальных условий состояния сети)

— Работоспособность приложения на различных устройствах разных производителей

### Тестирование производительности

1. Время загрузки приложения  
2. Обработка запросов  
3. Кэширование данных  
4. Потребление ресурсов приложением (например расход заряда батареи)

***Клиент-Серверная архитектура***

Термин «клиент-серверная архитектура» – сборное понятие, состоящее из двух взаимодополняющих компонентов: сервера и, собственно, клиента.

**Клиент** – локальный компьютер 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!

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.

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

**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).

**FTP** – протокол передачи информации из особого файлового сервера на ПК конечного пользователя.

**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.

ЕСТЬ СРАЗУ 2 ВИДА КЛИЕНТ-СЕРВЕРНЫХ АРХИТЕКТУР:

* Двухуровневая архитектура — это архитектура базы данных, в которой уровень представления выполняется на клиенте, а данные хранятся на сервере.
* Трехуровневая архитектура состоит из уровня представления (ПК, планшет, мобильный телефон и т. Д.), Уровня приложений (сервер) и сервера базы данных.

***Адаптивность vs Респонсивность в дизайне***

Сайты, оптимизированные под мобильные телефоны, не всегда создаются специально *для*мобильного устройства. Чаще всего это версии декстопного сайта, который будет работать на различных устройствах. Итак, главное отличие: оптимизация — это «подгон» дескотопного сайта под разные форматы, респонсивность — изначально создание мобильного сайта.  
Респонсивный сайт изменяется в зависимости от того, с какого устройства его просматривают. Например, у десктопного сайта может быть разметка с несколькими колонками, и мобильные пользователи видят такой сайт целиком. Но респонсивный дизайн  предполагает использование разметки с одной колонкой, которая лучше видна на маленьком сайте.

***URI, URL, URN***

### Расшифровка аббревиатур

**URL** - Uniform Resource Locator (унифицированный **определитель местонахождения** ресурса)

**URN** - Unifrorm Resource Name (унифицированное **имя** ресурса)

**URI** - Uniform Resource Identifier (унифицированный **идентификатор** ресурса)

Внимание! Далее в мелочах кроется истина, и пока ничего не понятно, - какая-то каша, но, едем дальше.

### В чем различия

**URL**: Исторически возник самым первым из понятий и закрепился как синоним термина веб-адрес. URL определяет местонахождение ресурса в сети и способ его (ресурса) извлечения.  
Это позволяет нам полностью узнать: как, кому и где можно достать требуемый ресурс, вводя понятия схемы, данных авторизации и местонахождения.

**URN**: Неизменяемая последовательность символов определяющая только имя некоторого ресурса. Смысл URN в том, что им единоразово и уникально именуется какая-либо сущность в рамках конкретного пространства имен (контекста), либо без пространства имен, в общем (что не желательно). Таким образом, URN способен преодолеть недостаток URL связанный с возможным будущим изменением и перемещением ссылок, однако, теперь для того, чтобы знать местонахождение URN ресурса необходимо обращаться к системе разрешения имен URN, в которой он должен быть зарегистрирован.

**URI**: Это лишь обобщенное понятие (множество) идентификации ресурса, включающее в нашем случае как URL, так и URN, как по отдельности, так и совместно. Т.е. мы можем считать, что: URI = URL или URI = URN или URI = URL + URN

### Подведем итоги

URI - это абстракция концепции идентификации,  
а URL и URN - это конкретные реализации - полного адреса ресурса и уникального контекстного имени соответственно.

***DNS***

Принцип работы DNS похож на поиск и вызов контактов из телефонной книги смартфона. Ищем имя, нажимаем «позвонить», и телефон соединяет нас с нужным абонентом. Понятно, что смартфон в ходе звонка не использует само имя человека, вызов возможен только по номеру телефона. Если вы внесете имя без номера телефона, позвонить человеку не сможете.

Так и с сайтом. Каждому имени сайта соответствует набор цифр формата 000.000.000.000. Этот набор называется IP-адресом, примером реального IP-адреса является 192.168.0.154 или 203.113.89.134. Когда пользователь вводит в адресной строке браузера имя сайта, например google.com, компьютер запрашивает IP-адрес этого сайта на специальном DNS-сервере и после получения корректного ответа открывает сам сайт.

DNS-сервер — это специализированный компьютер (или группа), который хранит IP-адреса сайтов. Последние, в свою очередь, привязаны к именам сайтов и обрабатывает запросы пользователя. В интернете много DNS-серверов, они есть у каждого провайдера и обслуживают их пользователей.

Сначала несколько слов о кэшировании. Выяснять связь между названием сайта и его IP-адресом требуется при каждом обращении к этому веб-сайту. Если сайт, который вы хотите посетить, находится достаточно далеко, многочисленные запросы к далёкому первичному DNS-серверу могут отнять много времени и замедлить загрузку веб-страниц. Чтобы избежать задержек, ближайший к вашему компьютеру DNS-сервер (обычно находящийся у вашего интернет-провайдера), сохраняет сведения о ранее запрошенных IP-адресах, и при повторном обращении к тому же сайту он сообщит его адрес очень быстро, так как будет хранить его в своём кэше.

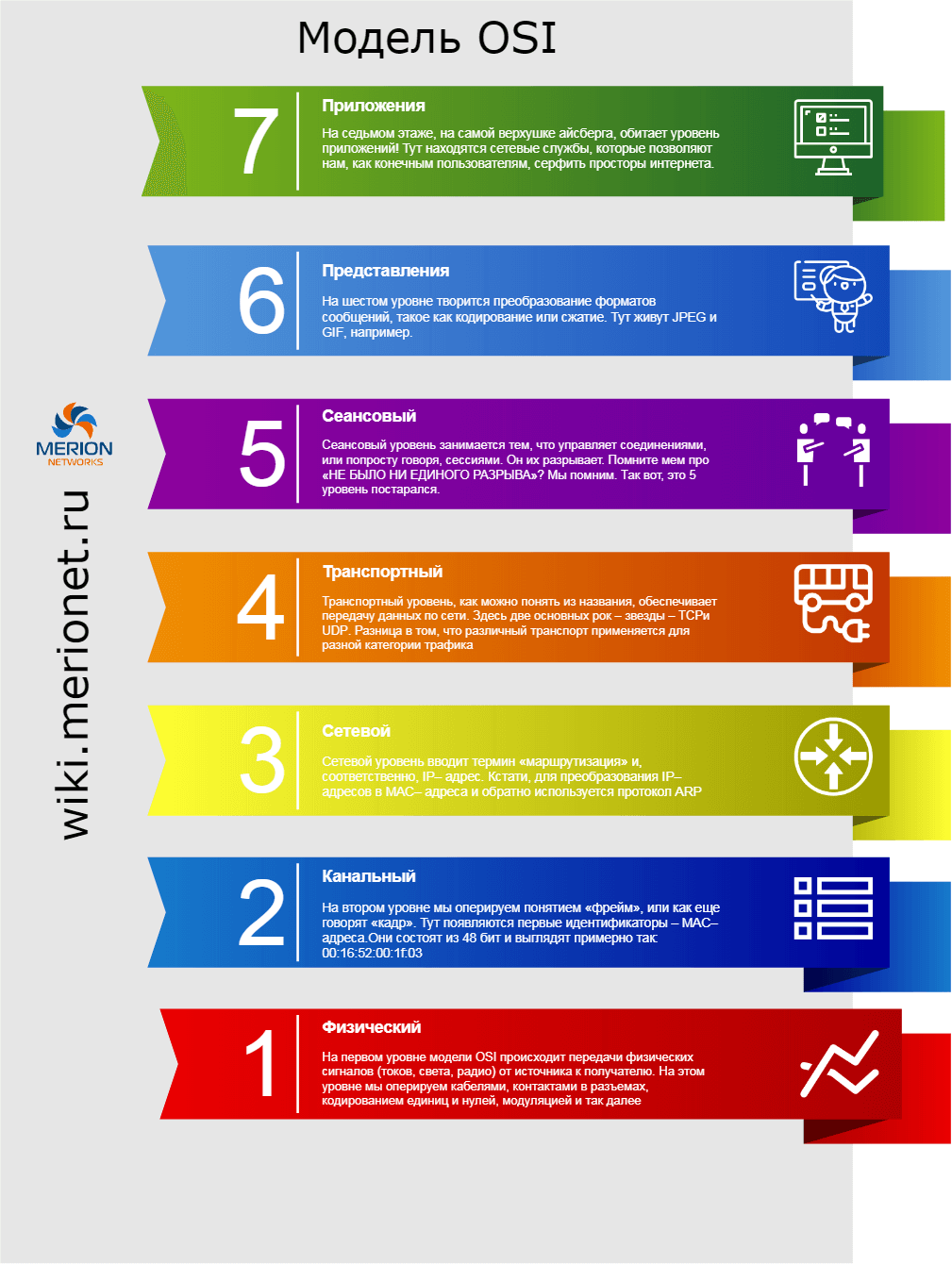
Но чтобы что-то кэшировать, нужно иметь источник кэшируемого. Таким источником служат первичные DNS-сервера, хранящие изначальные связи между доменами и их IP-адресами.

Для регистрации доменного имени достаточно его придумать. Но для того, чтобы оно начало «работать», вы должны сообщить регистратору доменное имя DNS-сервера, который будет хранить подробные данные о регистрируемом вами домене. Об этих данных будет сказано чуть позже.

Обычно используют два DNS-сервера: первичный и вторичный. Но их может быть и больше. Большее число DNS-серверов повышает надёжность доступа к вашему домену: если один окажется недоступен, ответит другой.

***OSI***

**Open Systems Interconnection (OSI)** – это скелет, фундамент и база всех сетевых сущностей. Модель определяет сетевые протоколы, распределяя их на 7 логических уровней. Важно отметить, что в любом процессе, управление сетевой передачей переходит от уровня к уровню, последовательно подключая протоколы на каждом из уровней.



***REST***

REST является стилем архитектуры ПО для систем распределения (пример - всем известная www). Обычно с его помощью строят веб-службы. **Рой Филдинг**, который имеет прямое отношение к созданию протокола HTTP, ввел термин **REST** в начале века. После этого любую систему, которая поддерживала его, стали называть **RESTFul**. Доступный интерфейс контроля без ненужных слоев - вот что из себя представляет эта архитектура. Глобальный ID, а именно URL, определяет каждую очередную информационную единицу. Сам же URL характеризуется форматом, который находится в жестких рамках.

#### REST НА ПРАКТИКЕ

Если нет пустых прослоек, данные будут переданы в виде, аналогичном им самим. "Заворачивание" информации в XML не происходит, как в случае с SOAP и XML-RPC, также не используется и AMF, как это бывает с Flash. По сути, происходит чистая передача.

**URL**, на деле, является базовым ключом для единицы данных. Пример: вторая книга из стопки, лежащей на столе, будет интерпретирована как /book/2, а 145 страница в этой книге - /book/2/page/145. Это достаточно жесткий формат. Самое интересное, что он может присутствовать в **любом** внешнем источнике, будь то **HTML** или **doc-файл**.

##### КОНТРОЛЬ ИНФОРМАЦИИ СЕРВИСА

Протокол передачи данных - основа контроля и управления. Самый часто используемый протокол - это, само собой, HTTP, и для него команды действий выглядят так:

* **GET** (получение);
* **PUT** (добавление);
* **POST** (изменение);
* **DELETE** (удаление).

Поэтому **Create/Read/Update/Delete**-действия будут выполнены и с 4 указанными алгоритмами, и посредством **GET** и **POST**. Это позволит в некоторых случаях обойти негативные эффекты с использованием непринятых **PUT** и **DELETE**.

###### REST В ПОСТРОЕНИИ ВЕБ-СЕРВИСОВ

Веб-сервис - приложение, которое работает в www с доступом, предоставленным через HTTP-протокол. Информационный обмен в рамках веб-сервиса осуществляется посредством формата XML. А это значит, что все данные из тела запросов будут всегда в аналогичном формате.

Для любой информационной единицы можно задать 5 вариантов действия:

* GET **/info/** (Index) используется для получения списка объектов. Конечно, полученный список будет ограничен заданным идентификатором.
* GET **/info/{id}** (View) позволяет получить всю информацию об объекте.
* PUT **/info/** или POST **/info/** (Create) осуществляет создание нового объекта.
* POST **/info/{id}** или PUT **/info/{id}** (Edit) производит замену данных, соответствующих введенному идентификатору.
* DELETE **/info/{id}** (Delete) удаляет данные, выделенные идентификатором.

###### ИТОГИ

Очевидно, что **REST**, как архитектура, обладает интуитивными алгоритмами и отличается простотой в использовании. Если запрос получен, то определить, что именно он делает, можно немедленно, без форматных разбирательств. При передаче не используются дополнительные слои, что обеспечивает REST особую ресурсоемкость.

###### ДЛЯ ЧЕГО МОЖНО ИСПОЛЬЗОВАТЬ?

Основное достоинство сервисов REST - дружественность. Рабочая связка с сайтом, flash, программой? Не важно, ведь алгоритмы парсинга XML и обработка запросов HTTP есть буквально везде. REST значительно упрощает все связанные с этим задачи. Конечно, на практике описанных здесь алгоритмов недостаточно, так как отсутствует защита данных. Но ввести в систему авторизацию и аутентификацию всегда можно при помощи **HTTP Authentication**.

***API***

**API (Application Programming Interface)** представляет собой совокупность различных инструментов, функций, реализованных в виде интерфейса для создания новых приложений, благодаря которому одна программа будет взаимодействовать с другой.

Основной задачей создания API была дать возможность программистам существенно облегчить задачу при разработке различных приложений за счет использования уже **готового кода** (какой-либо стандартной функции, процедуры, структуры или постоянного значения, которые будут в последующем выполнятся в конечном продукте).

API определяет возможную функциональность, которую определенная программа в форме библиотеки или модуля сможет осуществлять, при этом API позволяет **абстрагироваться** от способа реализации функционала.

#### ИСПОЛЬЗОВАНИЕ API В НАШЕ ВРЕМЯ

1. Хорошим примером использования API может служить процесс быстрой регистрации в различных приложениях используя аккаунт любой из предложенной социальной сети, когда посредствам специального API социальной сети (например, **Вконтакте**, **Facebook**) сторонние компании получают возможность использовать специальный код и API для предоставления Вам оперативного и упрощенного доступа к их продукту.
2. Компания **Google** за счёт использования **API** дает потенциальную возможность разработчикам различных приложений использовать интеграцию информации из своих сервисов на своих платформах. Благодаря этому вы сможете просмотреть видео, взятое из видео хостинга **YouTube.com** прямо внутри приложения.
3. Большое количество коммерческих компании предлагают API в качестве уже готового к использованию продукта. Так, американская компания **Weather Underground** зарабатывает за счёт продажи доступа к своему API для оперативного получения метеорологических данных в любой точке нашей планеты.

В заключении статьи следует отметить то, что в наше время невозможно создать качественные и полезные сервисы без использования библиотек API поскольку они необходимы как программистам для написания программного обеспечения и приложений, так и различным сервисам для предоставления услуг по обслуживанию клиентов и с каждым годом роль и область применения API только расширяется.

***SOAP REST***

Если совсем простыми, то некий сервис даёт тебе доступ к своим данным, но не напрямую, а через оболочку, которая назвается REST API. Тебе даётся некий url, по которому ты можешь получить те самые данные. Как правило, при изменении параметров этого url, ты получаешь разные данные.  
Пример:  
Допустим ты берешь в качестве сервиса Википедию. Его специальный url, по которому ты можешь получить данные (его еще называют endpoint) выглядит так: [**https://en.wikipedia.org/w/api.php**](https://en.wikipedia.org/w/api.php)  
Допустим нам нужно найти какую-то краткую выдержку из вики по ключевым словам. Первым делом мы идем в документацию и смотрим, какие параметры нам для этого нужны. Для данной задачи нам потребуется url со следующими параметрами:

https://en.wikipedia.org/w/api.php?action=opensearch&search=<search string>&prop=info&format=<format>&inprop=url

search string - словосочетание, по которому мы хотим искать.  
format - формат в котором будет выдан ответ (самые популярные это json и xml)  
  
Пример обращения к WikiAPI:

https://en.wikipedia.org/w/api.php?action=opensearch&search=microsoft&prop=info&format=json&inprop=url

Ответ от WikiAPI:

[

"microsoft",

[

"Microsoft",

"Microsoft Windows",

"Microsoft Word",

"Microsoft Excel",

"Microsoft Office",

"Microsoft Visual Studio",

"Microsoft PowerPoint",

"Microsoft SQL Server",

"Microsoft Office 2007",

"Microsoft Access"

1. **SOAP – это целое семейство протоколов и стандартов**, откуда напрямую вытекает, что это более тяжеловесный и сложный вариант с точки зрения машинной обработки. Поэтому REST работает быстрее.
2. SOAP используют HTTP как транспортный протокол, в то время как REST базируется на нем. Это означает, что все существующие наработки на базе протокола HTTP, такие как кеширование на уровне сервера, масштабирование, продолжают так же работать в REST архитектуре, а для SOAP необходимо искать другие средства. Взамен этого SOAP сервисы получают такое мифическое свойство, как возможность работать с любым протоколом транспортного уровня вместо HTTP, однако практической пользы от него зачастую не больше, чем сотрудникам [Челябинского трубопрокатного завода](http://ru.wikipedia.org/wiki/%D0%A7%D0%B5%D0%BB%D1%8F%D0%B1%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D1%80%D1%83%D0%B1%D0%BE%D0%BF%D1%80%D0%BE%D0%BA%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%B7%D0%B0%D0%B2%D0%BE%D0%B4) от большого количесва статей в википедиях на [мертвых языках](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%80%D1%82%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA).
3. Есть [мнение](http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest), что разработка RESTful сервисов намного проще. Наверное, это правда, если использовать Notepad в качестве основной среды разработки, но вот с использованием наших чудесных средств разработки, я позволю себе усомниться в верности этого утверждения.
4. В первом гугловском [результате](http://spf13.com/post/soap-vs-rest) по запросу «REST vs SOAP» акцентируется внимание на том, что ответ REST может быть представлен в различных форматах, а SOAP привязан к XML. Это действительно важный фактор, достаточно представить себе вызов сервиса из javascript, ответ на который мы определенно хотим получать в JSON.
5. **«REST vs SOAP» можно перефразировать в «Простота vs Стандарты»**, что проявляется в том, что для SOAP мы имеем протокол WSDL для исчерпывающего описания веб-сервиса, который с использованием все тех же чудесных средств разработки прото-таки волшебным образом делает почти всю работу за нас. Со стороны REST мы имеем загадочный и неиспользуемый протокол WADL, который, в принципе, и не нужен – он мешает простоте.
6. Второй аспект предыдущего пункта – обработка ошибок. В SOAP она полностью стандартизована, а REST может использовать давно известные коды ошибок HTTP (если здесь Вас посетила мысль, что это же очевидно и зачем я это пишу, то значит Вы внимательно читаете статью).
7. То, с чего можно было бы начать, но я припас напоследок. Это одна из ключевых мыслей. **SOAP работает с операциями, а REST – с ресурсами**. Этот факт в совокупности с отсутствием клиентского состояния у RESTful сервисов приводит нас к тому, что такие вещи как транзакции или другая сложная логика должна реализовываться «SOAP-но».

***Веб-сервис*** - это технология для взаимодействия между системами. REST подход подразумевает передачу представления ресурсов, это может быть HTML,JSP,XML.

Прежде всего, веб-сервисы (или веб-службы) — это технология. И как и любая другая технология, они имеют довольно четко очерченную среду применения.  
Если посмотреть на веб-сервисы в разрезе стека сетевых протококолов, мы увидим, что это, в классическом случае, не что иное, как еще одна надстройка поверх протокола HTTP.  
  
С другой стороны, если гипотетически разделить Интернет на несколько слоев, мы сможем выделить, как минимум, два концептуальных типа приложений — вычислительные узлы, которые реализуют нетривиальные функции и прикладные веб-ресурсы. При этом вторые, зачастую заинтересованы в услугах первых.  
Но и сам Интернет — разнороден, т. е. различные приложения на различных узлах сети функционируют на разных аппаратно-программных платформах, и используют различные технологии и языки.  
Чтобы связать все это и предоставить возможность одним приложениям обмениваться данными с другими, и были придуманы веб-сервисы.  
По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.  
Именно с появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

## Протоколы веб-сервисов На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

* **SOAP**(Simple Object Access Protocol) — по сути это тройка стандартов SOAP/WSDL/UDDI
* **REST**(Representational State Transfer)
* **XML-RPC** (XML Remote Procedure Call)

***SCRUM KANBAN***

### Две популярные Agile-методологии

**Scrum и Kanban** — представители методологий Agile-семейства. Обе считаются гибкими и итеративными. Перед тем, как разобраться в разнице между ними, вспомним кратко о том, что их объединяет.  
  
Более 17 лет назад лидеры IT-разработки сформулировали манифест Agile. Главное, что можем выделить из манифеста:

* Люди и взаимодействие важнее процессов и инструментов.
* Работающий продукт важнее исчерпывающей документации.
* Сотрудничество с заказчиком важнее согласования условий контракта.
* Готовность к изменениям важнее следования первоначальному плану.

## В чем разница между Scrum и Kanban?

Основу Scrum составляют короткие итерации или спринты, как правило, 2-3-х недельные. Перед началом спринта команда сама формирует список фич на итерацию, далее запускается спринт.  
  
После окончания спринта выполненные фичи заливаются на продакшн, а невыполненные — переносятся в другой спринт. Как правило, фичи, которые делаются во время спринта, не меняются: что было на старте спринта — должно быть сделано любой ценой к окончанию спринта.  
  
На Kanban мы посмотрим там, где он и возник. Представьте себе конвейер, на котором делают детали для машин Toyota. Есть станок, он делает зеркала для машин. Он умеет делать левые зеркала, правые зеркала, задние и зеркала для солнцезащитного козырька. Принцип прост: нажми на кнопку, поменяй режим — получи новую продукцию.  
  
Вот вы заказываете в Москве на Кутузовском новую Toyota Camry на «максималке», и для вас уже делают зеркала в козырьке (вы выбрали «максималку» как раз из-за зеркал в козырьке). Важный момент тут — мы можем менять приоритеты в любой момент. Мы очень быстро можем переключать станок в другой режим.  
  
**Основная разница между Scrum и Канбан — в длине итераций. В Scrum итерации — 2 недели, в Kanban задачи программисту можно «подсовывать» хоть каждый день.**  
  
Kanban дает больше гибкости, если под гибкостью понимать частоту смены приоритетов. Вчера вы залили на прод новую фичу, а сегодня получили данные с передовой и узнали, что вот эта штука не работает так, как было задумано — люди не нажимают кнопку «купить». Вы «даете по шапке» UX, он дает вам новые требования. Вы поднимаете наверх очереди эту задачу, программист берет эту задачу «сверху», выполняет ее и, к вечеру fix уже на проде, конверсия в платежи выросли на 12%. Это победа.  
  
В Scrum задачи принято оценивать в Story points или в часах. Без оценки не получится сформировать спринт: ведь нам нужно знать, успеем ли мы сделать задачи за 2 недели. Через 2 недели мы получаем ценную статистику — сколько часов или Story points команда смогла сделать за спринт. Velocity — это производительность команды за один спринт. Этот параметр позволяет Scrum менеджеру предсказать, где команда будет через 2 недели.  
  
В Kanban не принято делать оценку. Это опционально, команда решает сама. Здесь нет понятия «скорость работы команды», считается только среднее время на задачу. Время это считается с помощью специального отчета — Cycle Time.

Итак, в Scrum наша цель — закончить спринт, в Kanban — задачу.

***SCRUM TEAM***

**Владелец продукта (Product Owner)**

Владелец продукта отвечает за достижение максимальной ценности продукта.

**Владелец продукта**— единственный человек, отвечающий за список требований к продукту и ответственный за результат работы команды. Роль объединяет в себе функции разных ролей спонсора, менеджера продукта, руководителя проекта, менеджера по маркетингу, представителя клиента, участника команды. Всегда доступен команде, чтобы разъяснить требования или принять необходимые решения. Но не имеет формальной власти над командой разработки, является частью общей scrum-команды и должен быть не только лидером, но и хорошим командным игроком.

**Команда создания продукта (Development Team)**

Один в поле не воин, нужна команда. **Команда создания продукта** — самоорганизующаяся, кросс-функциональная команда, которая на выходе каждой итерации создает потенциально продаваемый вариант продукта. Размер команды — от 3 до 9 человек. Для команды меньше трех человек нет смысла вводить какие-то практики. В командах больше 9 человек будут сложности с координацией и взаимодействием.

Внутри команды есть только одна роль — developer. Других должностей, ролей, под-команд — не должно быть.

Почему так и кто такой Developer?

Мы часто ставим знак равенства: developer = разработчик = программист. Это не совсем верно. Более широкое понимание слова developer, которое, например, приведено в словаре MacMillan- “someone whose job is to create new ideas and products” — “некто, чьей работой является создание новых идей и продуктов”. В понимании Скрам **developer** — один из разработчиков или, лучше , создателей продукта. На мой взгляд, Development Team — лучше переводить как “команда создания продукта”, а не “команда разработки продукта”. Чтобы убрать неправильную ассоциацию с программистами.

**Кросс-функциональная команда** — в команде должны быть все знания и навыки, позволяющие создать готовый к продаже продукт. Если говорить о программном продукте, то это дизайнеры, аналитики, программисты, тестировщики, devops-инженеры и так далее.

**Скрам-мастер (Scrum Master)**

Для этого появляется новая, многим непонятная роль скрам-мастера. Основная ответственность скрам-мастера — организовать эффективный процесс. Скрам-мастер и владелец продукта дополняют друг друга. Владелец продукта нацеливается на создание лучшего продукта, скрам-мастер — на создание правильного процесса. Правильный процесс приводит к правильному результату. Сделать из индивидов самоорганизующуюся команду, способную эффективно решать инновационные задачи с высокой степенью неопределенности.

Скрам-мастер, как и владелец продукта, должен совмещать в себе одновременно много ролей.

* обслуживающий лидер (servant leader) — помогает понять и внедрить процессы Скрам, помогает повысить прозрачность, доверие, устранять препятствия
* коуч — помогает раскрыть потенциал отдельных людей и команды, помогает в решении конфликтов
* фасилитатор — помогает группе определить общие цели и достигать их, оставаясь нейтральным
* учитель — помогает изучать новые вещи, как лучше и эффективнее организовывать работу
* ментор — обладает глубокими знаниями и может помочь советом опытного товарища
* менеджер — устраняет препятствия, помогает развитию команды, способствует развитию правильной культуры команды
* агент изменений — помогает проводить изменения к

***Интернационализация, локализация (Глобалтзация)***

Ни для кого не секрет, что успех приложений на мировом рынке сегодня во многом зависит от того, насколько удачно приложение адаптировано для работы по всему миру. Стратегия по подготовке приложения к интернациональному использованию называется **глобализацией**.

Если компанией-разработчиком продукта взят курс на глобализацию, то необходимо запланировать интернационализацию, локализацию продукта, а также перевод контента ПО на целевые языки.

**Интернационализация** (I18n) — процесс создания и разработки продукта, который в последующем обеспечит безбарьерную локализацию ПО. Например, разработчик должен заложить в CSS основу для вертикального текста, если в будущем планируется локализовать продукт на язык с вертикальным письмом.

**Локализация** (L10n) – адаптация продукта и его документации к языковым соответствиям, культурным и другим требованиям целевого рынка или рынков

Переходя непосредственно к задачам I18n и L10n, выделим следующие пункты, на которые должен обратить внимание специалист по тестированию:  

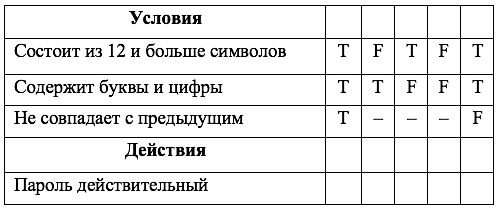

***Прокси-сервер*** (от англ. *proxy* — представитель, уполномоченный; часто просто *прокси*, *сервер-посредник*) — промежуточный сервер (комплекс программ) в компьютерных сетях, выполняющий роль посредника между пользователем и целевым сервером (при этом о посредничестве могут как знать, так и не знать обе стороны), позволяющий клиентам как выполнять косвенные запросы (принимая и передавая их через прокси-сервер) к другим сетевым службам, так и получать ответы. Сначала клиент подключается к прокси-серверу и запрашивает какой-либо ресурс (например [e-mail](https://ru.wikipedia.org/wiki/E-mail)), расположенный на другом сервере. Затем прокси-сервер либо подключается к указанному серверу и получает ресурс у него, либо возвращает ресурс из собственного [кэша](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88) (в случаях, если прокси имеет свой кэш). В некоторых случаях запрос клиента или ответ сервера может быть изменён прокси-сервером в определённых целях. Прокси-сервер позволяет защищать компьютер клиента от некоторых сетевых атак и помогает сохранять анонимность клиента, но также может использоваться мошенниками для скрытия адреса сайта, уличённого в мошенничестве, изменения содержимого целевого сайта (подмена), а также перехвата запросов самого пользователя.

***TCP/IP VS OSI***

***Отличие PUT от PATCH***

PUT - обновление объекта целиком, PATCH - обновление поля объекта, можно и методом PUT обновить одно поле, однако метод PUT будет **проходить все поля** объекта и искать необходимое, в отличии от PATCH, который **не обходит объект** целиком.

***DECISION TABLE***



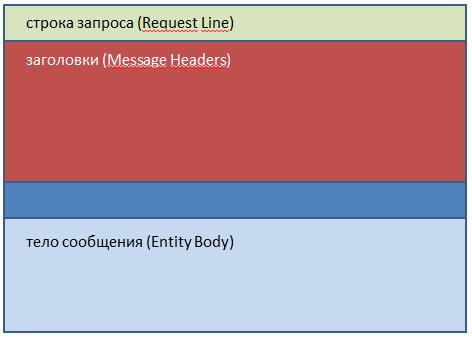
GET тоже может передавать информацию, но делает это не секьюрно.

**Запрос GET передает данные в URL** в виде пар "имя-значение" (другими словами, через ссылку), а **запрос POST передает данные в теле запроса** (подробно показано в примерах ниже). Это различие определяет свойства методов и ситуации, подходящие для использования того или иного HTTP метода.

***Структура HTTP запросов и ответов.***

Давайте более подробно рассмотрим эту структуру, по которой строятся запросы и ответы в протоколе HTTP.

HTTP запрос состоит из трех основных частей, которые идут в нем именно в том порядке, который указан ниже. Между заголовками и телом сообщения находится пустая строка (в качестве разделителя), она представляет собой символ перевода строки.



1. строка запроса (Request Line)

2. заголовки (Message Headers)

Пустая строка (разделитель)

3. тело сообщения (Entity Body) – необязательный параметр

**Строка запроса** – указывает метод передачи, URL-адрес, к которому нужно обратиться и версию протокола HTTP.

**Заголовки** – описывают тело сообщений, передают различные параметры и др. сведения и информацию.

**тело сообщения**  - это сами данные, которые передаются в запросе.  Тело сообщения – это необязательный параметр и может отсутствовать.

Когда мы получаем ответный запрос от сервера, тело сообщения, чаще всего представляет собой содержимое веб-страницы. Но, при запросах к серверу, оно тоже может иногда присутствовать, например, когда мы передаем данные, которые заполнили в форме обратной связи на сервер.

Более подробно, каждый элемент запроса, мы рассмотрим в следующих заметках.

Давайте, для примера, рассмотрим один реальный запрос к серверу. Я выделил каждую часть запроса своим цветом: строка запроса - зеленый, заголовки - оранжевый, тело сообщения- голубой.

*Запрос от браузера:*

GET / HTTP/1.1

Host: webkyrs.info

User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:18.0) Gecko/20100101 Firefox/18.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Cookie: wp-settings

Connection: keep-alive

В следующем примере уже присутствует тело сообщения.

*Ответ сервера:*

HTTP/1.1 200 OK

Date: Sun, 10 Feb 2013 03:51:41 GMT

Content-Type: text/html; charset=UTF-8

Transfer-Encoding: chunked

Connection: keep-alive

Keep-Alive: timeout=5

Server: Apache

X-Pingback: //webkyrs.info/xmlrpc.php

<!DOCTYPE html PUBLIC «-//W3C//DTD XHTML 1.0 Transitional//EN» «http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd»>

<html xmlns=»http://www.w3.org/1999/xhtml»>

<head>

<meta http-equiv=»Content-Type» content=»text/html; charset=utf-8″ />

<title>Документ без названия</title>

</head>

<body>

</body>

</html>

Вот такими сообщениями обмениваются клиент и сервер по протоколу HTTP.

**Тестирование программного обеспечения (Software Testing)** — проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определённым образом.

**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 обнаружение 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.

**Обеспечение качества (QA — Quality Assurance)** и **контроль качества (QC — Quality Control)** **QA** (сокращённо от «**Quality Assurance**», что с английского переводится, как «**обеспечение качества**») это превентивный процесс, гарантирующий, что все необходимые техники, процедуры, стандарты и методологии соблюдаются в процессе разработки продукта для предоставления результата без дефектов.

**QC** (сокращенно от «**Quality Control**», что с английского переводится, как «**контроль качества**») означает процесс проверки качества, который обеспечивает соответствие продукта ранее установленным требованиям. Проще говоря, это процесс, который обеспечивает ожидаемое качество продукта.

В центре внимания QA будет процесс предотвращения дефектов разрабатываемого продукта, в то время как QC будет сосредоточен на конечном качестве продукта и устранении недостатков.

|  |  |
| --- | --- |
| **Quality Assurance**  QA – это проактивный процесс, направленный на предотвращение возможных дефектов. Он выполняется во время разработки продукта. | **Quality Control**  QC – это реактивный процесс, целью которого является подтверждение качества конкретного продукта посредством тестирования, выявления и устранения неисправностей. Он проводится после того, как продукт был разработан. |
| **Quality Assurance**  **На процесс** | **Quality Control**  **На продукт** |

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

**Валидация (validation)** — это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, его требованиям к системе.

С помощью **валидации**Вы можете быть уверенным в том, что создали «правильный» продукт. Продукт, который полностью удовлетворяет заказчика.

С помощью **верификации**Вы можете увериться в том, что продукт сделан «правильно»: придерживаясь необходимых методик, инструментов и стандартов.

Документацию, которая используется на проектах по разработке ПО, можно условно разделить на две группы:

1. Проектная документация — включает в себя всё, что относится к проекту в целом.
2. Продуктовая документация — часть проектной документации, выделяемая отдельно, которая относится непосредственно к разрабатываемому приложению или системе.

**Этапы тестирования:**

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

**Стадии разработки ПО** — этапы, которые проходят команды разработчиков ПО, прежде чем программа станет доступной для широкого круга пользователей.  
  
**Программный продукт проходит следующие стадии:**

1. анализ требований к проекту;
2. проектирование;
3. реализация;
4. тестирование продукта;
5. внедрение и поддержка.

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

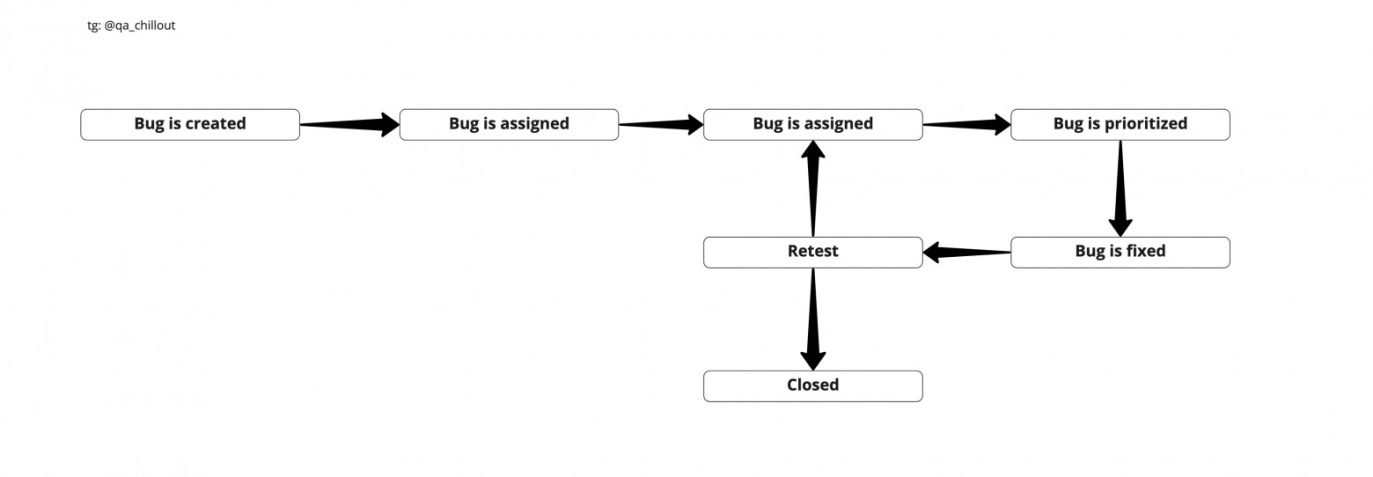
1. **Корректность** — точное описание разрабатываемого функционала.
2. **Проверяемость** — формулировка требований таким образом, чтобы можно было выставить однозначный вердикт, выполнено все в соответствии с требованиями или нет.
3. **Полнота** — в требовании должна содержаться вся необходимая для реализации функциональности информация.
4. **Недвусмысленность** — требование должно содержать однозначные формулировки.
5. **Непротиворечивость** — требование не должно содержать внутренних противоречий и противоречий другим требованиям и документам.
6. **Приоритетность** — у каждого требования должен быть приоритет(количественная оценка степени значимости требования). Этот атрибут позволит грамотно управлять ресурсами на проекте.
7. **Атомарность** — требование нельзя разбить на отдельные части без потери деталей.
8. **Модифицируемость** — в каждое требование можно внести изменение.
9. **Прослеживаемость** — каждое требование должно иметь уникальный идентификатор, по которому на него можно сослаться.

**Дефект (bug)** — отклонение фактического результата от ожидаемого.

**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.  
  
**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. Вложения (Attachments) — скриншоты, видео или лог-файлы.
8. 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. Окружение (Environment) – окружение, на котором воспроизвелся баг.

**Жизненный цикл бага**

[](https://habrastorage.org/webt/dg/yy/w9/dgyyw9t6waodwooi6pnis6w2wai.jpeg)

**Severity vs Priority**

**Серьёзность (severity)** показывает степень ущерба, который наносится проекту существованием дефекта. Severity выставляется тестировщиком.  
  
**Градация 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.

**Срочность (priority)** показывает, как быстро дефект должен быть устранён. Priority выставляется менеджером, тимлидом или заказчиком  
  
**Градация Приоритета дефекта (Priority):**

* **P1 Высокий (High)**  
  Критическая для проекта ошибка. Должна быть исправлена как можно быстрее.
* **P2 Средний (Medium)**  
  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.

**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.
* **Под-задача (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.

**Тест-дизайн** — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы).  
  
**Техники тест-дизайна**  
  
Автор книги "[A Practitioner’s Guide to Software Test Design](http://testingbooks.ru/a-practitioners-guide-to-software-test-design/)", Lee Copeland, выделяет следующие 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) на группы эквивалентных по своему влиянию на систему значений.
2. **Техника анализа граничных значений (boundary value testing)** — это техника проверки поведения продукта на крайних (граничных) значениях входных данных.
3. **Попарное тестирование (pairwise testing)** — это техника формирования наборов тестовых данных из полного набора входных данных в системе, которая позволяет существенно сократить количество тест-кейсов.
4. **Тестирование на основе состояний и переходов (State-Transition Testing)** — применяется для фиксирования требований и описания дизайна приложения.
5. **Таблицы принятия решений (Decision Table Testing)** — техника тестирования, основанная на методе чёрного ящика, которая применяется для систем со сложной логикой.
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.
7. **Сценарий использования (Use Case Testing)** — Use Case описывает сценарий взаимодействия двух и более участников (как правило — пользователя и системы).

### Методы тестирования

[](https://habrastorage.org/webt/b_/np/bl/b_npblhfjf2w32kxtkuwnnft_-g.png)  
  
**Тестирование белого ящика** — метод тестирования ПО, который предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику.  
  
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.
* Почему «белый ящик»? Тестируемая программа для тестировщика — прозрачный ящик, содержимое которого он прекрасно видит.

**Тестирование серого ящика** — метод тестирования ПО, который предполагает комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично.

**Тестирование чёрного ящика** — также известное как тестирование, основанное на спецификации или тестирование поведения — техника тестирования, основанная на работе исключительно с внешними интерфейсами 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.

### Тестовая документация

**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.  
  
**Тест план должен отвечать на следующие вопросы:**

* Что необходимо протестировать?
* Как будет проводиться тестирование?
* Когда будет проводиться тестирование?
* Критерии начала тестирования.
* Критерии окончания тестирования.

**Основные пункты тест плана:**

1. Идентификатор тест плана (Test plan identifier);
2. Введение (Introduction);
3. Объект тестирования (Test items);
4. Функции, которые будут протестированы (Features to be tested;)
5. Функции, которые не будут протестированы (Features not to be tested);
6. Тестовые подходы (Approach);
7. Критерии прохождения тестирования (Item pass/fail criteria);
8. Критерии приостановления и возобновления тестирования (Suspension criteria and resumption requirements);
9. Результаты тестирования (Test deliverables);
10. Задачи тестирования (Testing tasks);
11. Ресурсы системы (Environmental needs);
12. Обязанности (Responsibilities);
13. Роли и ответственность (Staffing and training needs);
14. Расписание (Schedule);
15. Оценка рисков (Risks and contingencies);
16. Согласования (Approvals).

**Чек-лист (check list)** — это документ, который описывает что должно быть протестировано. Чек-лист может быть абсолютно разного уровня детализации

Чаще всего чек-лист содержит только действия, без ожидаемого результата. Чек-лист менее формализован.

**Тестовый сценарий (test case)** — это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

***Основные коды ответов HTTP:***

200-ok

201-created "Создано". Запрос успешно выполнен и в результате был создан ресурс.

301-moved permanently "Перемещён на постоянной основе". Этот код ответа значит, что URI запрашиваемого ресурса был изменён. Возможно, новый URI будет предоставлен в ответе.

400-Bad request "Плохой запрос". Этот ответ означает, что сервер не понимает запрос из-за неверного синтаксиса.

401-Unathorized "Неавторизованно". Для получения запрашиваемого ответа нужна аутентификация. Статус похож на статус 403, но,в этом случае, аутентификация возможна.

403-Forbidden "Запрещено". У клиента нет прав доступа к содержимому, поэтому сервер отказывается дать надлежащий ответ.

404-not found "Не найден". Сервер не может найти запрашиваемый ресурс. Код этого ответа, наверно, самый известный из-за частоты его появления в вебе.

500-Internal Server Error "Внутренняя ошибка сервера". Сервер столкнулся с ситуацией, которую он не знает как обработать.

502-Bad gateway "Плохой шлюз". Эта ошибка означает что сервер, во время работы в качестве шлюза для получения ответа, нужного для обработки запроса, получил недействительный (недопустимый) ответ.

503-Service Unavailable "Сервис недоступен". Сервер не готов обрабатывать запрос. Зачастую причинами являются отключение сервера или то, что он перегружен.

***Отличие system testing от end-to-end testing.***

**System testing**: You're **testing the whole system** i.e. all of it's components to ensure that each is functioning as intended. This is more from a functional side to check against requirements.

**End to end testing**: This is more about the **actual flow *through* a system** in a more realistic end user scenario. Can a user navigate the application as expected and does it work. You're testing the workflow.

***SSO***

Что такое single sign-on?

Технология единого входа (Single sign-on SSO) — метод аутентификации, который позволяет пользователям безопасно аутентифицироваться сразу в нескольких приложениях и сайтах, используя один набор учетных данных.

**Как работает SSO?**

SSO базируется на настройке доверительных отношений между приложением, известным как провайдер услуг, и системой управления доступами, например, OneLogin. Такие доверительные отношения часто базируются на обмене сертификатом между системой управления доступами и провайдером услуг. Такой сертификат может использоваться, чтобы обозначить идентификационную информацию, которая отправляется от системы управления доступами провайдеру услуг, таким образом провайдер услуг будет знать, что информация поступает из надежного источника. В SSO идентификационные данные принимают форму токенов, содержащих идентификационные значения информации о пользователе такие, как email или имя пользователя.

***Уровни и Типы Требований к ПО***

**Уровни требований:**

***Уровень Бизнес-требований***

***Уровень Пользовательских требований***

***Уровень Продуктных требований***

**Бизнес-требования (Business Requirements)**через них выражается цель, ради которой создается продукт (для чего продукт, какая от продукта польза, как будем получать прибыль).

Результатом сформированных требований является:

**Общее видение** (**Vision**and **Scope**) - это документ с простым текстом и таблицами, в нем нет деталей и технических характеристик.

Пример бизнес-требований:

Нужен инструмент, в реальном времени отражающий наиболее выгодный курс покупки и продажи валюты

Необходимо в два-три раза повысить количество заявок, обрабатываемых одним оператором за смену

Нужно автоматизировать процесс выписки товарно-транспортных накладных на основе договоров

**Пользовательские требования (User Requirements)**описывают задачи, которые пользователь может выполнять с помощью разрабатываемого продукта, а также способы (сценарии) их решения в системе (реакция продукта на действия пользователя, сценарии работы пользователя).

Пользовательские требования представлены в виде:

вариантов использования (uses cases)

пользовательских историй (user stories)

пользовательских сценариев (user scenarios)

**Бизнес-правила (Business Rules)** - описывают возможности принятых процессов, огарничений, правил.

**Продуктные требования:**

**Функциональные требования (Functional Requirements)** — охватывают предполагаемое поведение системы, определяя действия, которые система способна выполнять. Описывается в системной спецификации. В основном влияют на дизайн системы.

Пример функциональных требований:

В процессе инсталяции приложение должно проверять остаток свободного места на целевом носителе

Приложение не должно выгружать из памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции

**Нефункциональные требования (Non-functional Requirements) -**охватывают свойства системы (удобства использования, надежность, масштабируемость), которыми она должна обладать при реализации своего поведения.

Нефункциональные требования  в основоном влияют на архитектуру продукта.

Пример нефункциональных требований:

Вне зависимости от условий работы, общий объем используемой продуктом оперативной памяти не должен превышать 2ГБ

При одновременной непрерывной работе с системой 1000 юзеров, время между возникновением сбоев должно быть более или равно 200 часов.

***Источники требований***

* Федеральное и муниципальное отраслевое законодательство (конституция, законы, распоряжения)
* Нормативное обеспечение организации (регламенты, положения, уставы, приказы)
* Текущая организация деятельности объекта автоматизации
* Модели деятельности (диаграммы бизнес-процессов)
* Представления и ожидания потребителей и пользователей системы
* Журналы использования существующих программно-аппаратных систем
* Конкурирующие программные продукты

Методы выявления требований:

* Интервью, опросы, анкетирование
* Мозговой штурм, семинар
* Наблюдение за производственной деятельностью, «фотографирование» рабочего дня
* Анализ нормативной документации
* Анализ моделей деятельности
* Анализ конкурентных продуктов
* Анализ статистики использования предыдущих версий системы

***Браузер и браузерный движок***

Это программный модуль, который преобразовывает содержимое веб-страниц, это файлы в формате XML, HTML, а также цифровые изображения. Также преобразуется и информация о форматировании, которая обычно представлена в форматах XSL, CSS и других. Все это, после преобразования, представляется в интерактивном отображении содержимого на мониторе.

**Кто создает браузеры и движки**

С браузерными движками дела обстоят кардинально иначе. Вот список движков, их разработчиков и обладателей:

Blink: Google

Gecko: Mozilla

WebKit: Apple

EdgeHTML: Microsoft

Это основные обладатели и разработчики движков. В разработке им помогают сторонние компании на определенных условиях.

Например, рассмотрим самый популярный проект от компании Google — Chromium. Данный проект содержит в себе движок Blink. Создавать данный проект Google помогают такие компании как Intel, Facebook, IBM, LG Electronics, NVIDIA, Yandex. Полный список можно посмотреть [тут](https://chromium.googlesource.com/chromium/src.git/+/master/AUTHORS).  
Условия на которых компании помогают создавать Blink/Chromium описаны в разделе [Legal stuff](https://www.chromium.org/developers/contributing-code" \l "TOC-Legal-stuff) на сайте проекта Chromium. Если кратко, то всё что вы создаёте принадлежит (не эксклюзивно) Google.

***Кэш и куки***

**Кэш**

Кэш представляет собой промежуточный буфер памяти и место на жестком диске персонального компьютера, где хранится информация о ранее посещенных веб-страницах, включая изображения и прочие данные.

Представьте, что вы впервые зашли на какой-нибудь сайт. Вы просматриваете разные страницы, при этом вся информация (графическая, текстовая, мультимедийная) загружается на сайт с соответствующего сервера. Когда кэширование подключено, данные с посещенных вами страниц сохраняются на жестком диске в специально отведенной для этого папке.

Когда вы посещаете этот сайт второй и последующие разы, ваш веб-браузер проверяет содержимое сайта, а потом выполняет загрузку лишь новой информации. Остальная же информация берется для отображения именно из вашего кэша. Для чего? Во-первых, для повышения скорости загрузки сайта, во-вторых, для снижения нагрузки на интернет-соединение, ведь доступ к данным в кэше осуществляется быстрее.

**Куки**

Главное отличие куки от кэша заключается в том, что каждый раз, когда вы повторно заходите на конкретный сайт, с которого вам был когда-то отправлен конкретный куки, веб-клиент (обычно, это ваш веб-браузер) **пересылает этот фрагмент данных web-серверу в составе HTTP-запроса**.

Как правило, применение куки упрощает следующие процедуры: • аутентификацию пользователя; • хранение персональных настроек и предпочтений; • отслеживание состояния сеанса доступа; • ведение статистики и т. д.

Вот несколько примеров использования куки на практике: 1. **Авторизация на сайте**. Как известно большинство сайтов имеют авторизацию (ввод пароля, логина, телефона, почты и т. п.). Cookie могут применяться сервером для опознания ранее аутентифицированных пользователей. 2. **Корзина** в интернет-магазинах. Если не использовать куки, при выборе товара и переходе на новую страницу товар может исчезнуть. 3. **Настройки**. К примеру, вы выставили нужные настройки региона, языка и т. д. Без куки они могут сброситься и вернуться в статус значений по умолчанию.

***Отличие quality control от testing иquality assurance***

To summarize, Quality assurance is a set of processes that help “avoid” defects and assure quality. While Quality Control is a **set of** activities that help detect defects and quality issues before the products reach the hands of end customers. Testing is one of the ways of detecting those defects.

***Отличия веб сайта от веб приложения:***

**Web-сайты**

Обычно носят информационный характер. Состоят из веб-страниц, объединенных друг с другом в единый ресурс. Имеют простую архитектуру на основе HTML-кода. Служат в качестве платформы для предоставления контента для посетителей: могут содержать текст, изображения или музыку.

Сайты не предоставляют возможности взаимодействия с программой. Пользователи не имеют доступа к размещению своей информации кроме заполнения формы для получения подписки. Наиболее яркими примерами типичных сайтов являются новостные, кулинарные, прогнозы погоды.

**Web-приложения**

Это интерактивные компьютерные приложения, разработанные для сети интернет, позволяющие пользователям вводить, получать и манипулировать данными с помощью взаимодействия. Такие программы обычно имеют очень тесную связь с сервером, отправляя на него множество запросов. Могут быть встроены в web-страницы, либо сами web-страницы могут быть приложениями. Например, Facebook, Gmail, YouTube, Ebay, Twitter, Amazon.

Web-приложения используют имя пользователя и пароль для аутентификации. Позволяют своим посетителям обмениваться мгновенными сообщениями (чат-платформы, соцсети, блоги), создают контент на основе пользовательских предпочтений, обеспечивают к нему неограниченный доступ, используют мини-встроенные программы для развлечений.

Многие интернет-приложения могут не иметь реального информативного содержания. Они используются для выполнения дополнительных задач (интернет-переводчики, конвертеры файлов, мессенджеры).

Поскольку использование интернет-приложений часто связано с вводом личных данных, платежных реквизитов, разработчики несут ответственность за защиту конфиденциальной информации. Они сталкиваются с дополнительными требованиями, такими как соответствие стандарту безопасности PCI DSS.

**Основные отличия**

Веб-сайт является источником информации, в то время как веб-приложение работает в интерактивном режиме.

Функции, задачи, пользовательский интерфейс веб-приложения гораздо сложнее.

Веб-приложение может быть составной частью сайта либо отдельным ресурсом.

Разработка web-сайта является более легким, недорогим проектом. Для разработки web-приложения требуются большая вычислительная мощность, а также различные платформы и инструменты.

Веб-приложение является более ресурсоемким, так как может взаимодействовать с пользователем и выполнять различные действия.

## *Идентификация, аутентификация и авторизация*

Для начала система запрашивает логин, пользователь его указывает, система распознает его как существующий — это **идентификация**.

После этого Google просит ввести пароль, пользователь его вводит, и система соглашается, что пользователь, похоже, действительно настоящий, раз пароль совпал, — это [**аутентификация**](https://encyclopedia.kaspersky.ru/glossary/authentication/).

Скорее всего, Google дополнительно спросит еще и одноразовый код из SMS или приложения. Если пользователь и его правильно введет, то система окончательно согласится с тем, что он настоящий владелец аккаунта, — это [**двухфакторная аутентификация**](https://encyclopedia.kaspersky.ru/glossary/two-factor-authentication/).

После этого система предоставит пользователю право читать письма в его почтовом ящике и все в таком духе — это [**авторизация**](https://encyclopedia.kaspersky.ru/glossary/authorization/).

Аутентификация без предварительной идентификации лишена смысла — пока система не поймет, подлинность чего же надо проверять, совершенно бессмысленно начинать проверку. Для начала надо представиться.

Идентификация без аутентификации — это просто глупо. Потому что мало ли кто ввел существующий в системе логин! Системе обязательно надо удостовериться, что этот кто-то знает еще и пароль. Но пароль могли подсмотреть или подобрать, поэтому лучше подстраховаться и спросить что-то дополнительное, что может быть известно только данному пользователю: например, одноразовый код для подтверждения входа.

А вот авторизация без идентификации и тем более аутентификации очень даже возможна. Например, в Google Документах можно публиковать документы так, чтобы они были доступны вообще кому угодно. В этом случае вы как владелец файла увидите сверху надпись, гласящую, что его читает неопознанный енот. Несмотря на то, что енот совершенно неопознанный, система его все же авторизовала — то есть выдала право прочитать этот документ.

## *****Типы cookies*****

Cookies бывают разными… Различают 3 основных типа cookies:

* **сессионные:** они истекают сразу, как только Вы закрываете свой браузер (или если Вы остаетесь неактивны некоторое время). Они используются, например, сайтами онлайн-торговли, чтобы позволить Вам продолжать навигацию, не теряя того, что Вы положили в корзину.
* **постоянные:** они остаются, даже если браузер закрыт. У них есть срок действия, по закону - не дольше 6 месяцев. Они запоминают Ваши пароли, логины и прочую информацию, которую Вам не придется снова и снова вводить.
* **третьих лиц:** атрибуты cookies обычно соответствуют домену соответствующего веб-сайта. Но это недействительно для cookies третьих лиц, так как они устанавливаются (как Вы уже могли догадаться из названия) третьей стороной, например, рекламными сайтами. Они собирают данные о Ваших «браузерных» привычках, а потом позволяют отслеживать Вашу активность на многих сайтах. Примеры сайтов, использующих cookies третьих лиц: Facebook, Flickr, Google Analytics, Google Maps, Google Plus, SoundCloud, Tumblr, Twitter и YouTube.

***Activity Life Cycle***

Activity — это отдельный экран в Android. Это как окно в приложении для рабочего стола, или фрейм в программе на Java. Activity позволяет вам разместить все ваши компоненты пользовательского интерфейса или виджеты на этом экране.

Важно понимать, что у Activity есть жизненный цикл, проще говоря, это означает, что она может быть в одном из различных стадий, в зависимости от того, что происходит с приложением при действиях пользователя.

Create

Start

Resume

Pause

Stop

Destroy

***DOS VS DDOS (denial-of-service, distributed denial-of-service)***

The principal difference between a DoS and a DDoS is that the former is a system-on-system attack, while the latter involves several systems attacking a single system.

***UTM-метки*** – своего рода теги, которые добавляют в адресную строку сайта или страницы, они нужны для того, чтобы получить подробную информацию о каждом источнике трафика.

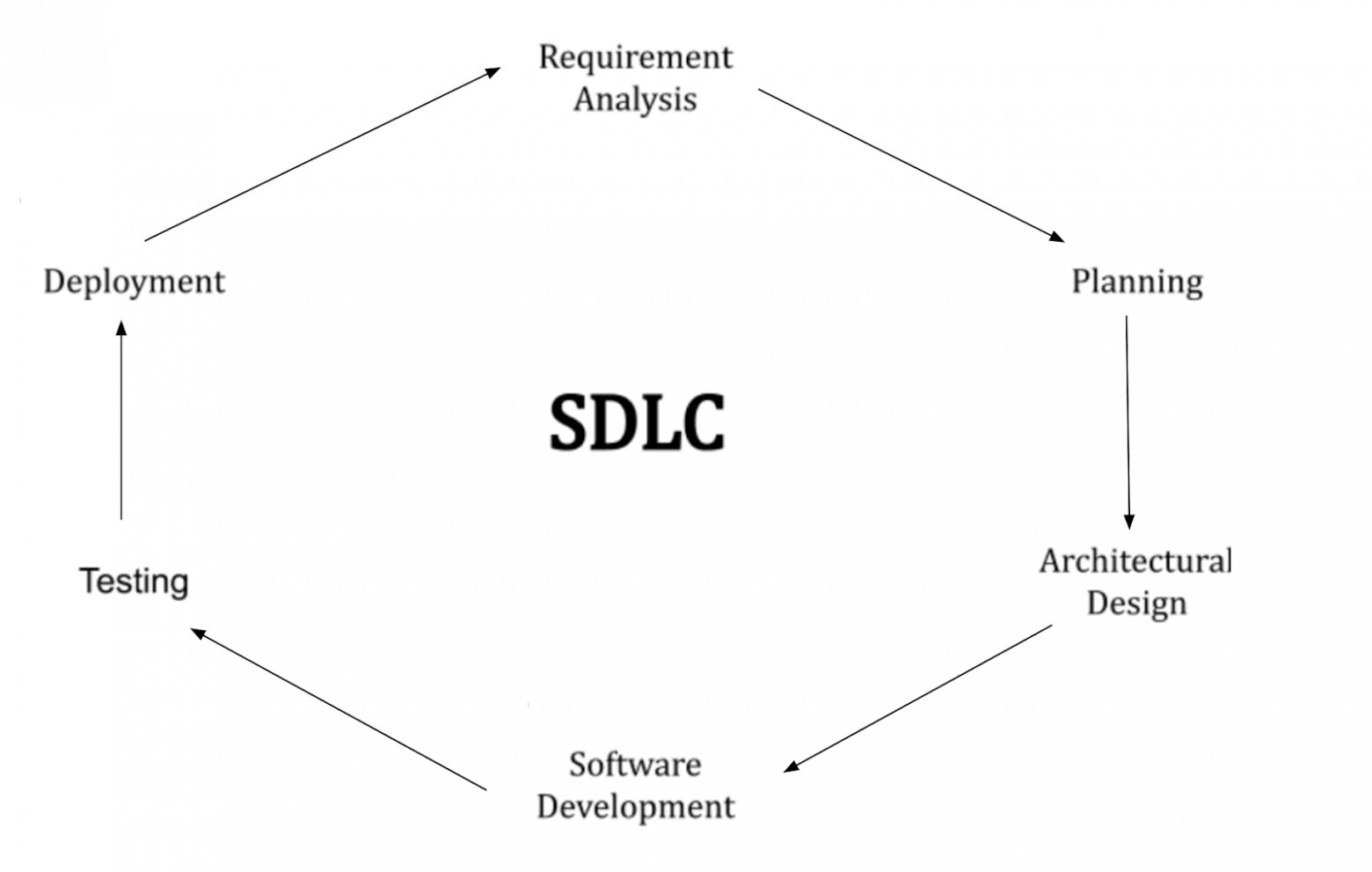
Их используют для отслеживания эффективности рекламной кампании или чтобы узнать о других источниках трафика. Большинство рекламщиков и маркетологов создают UTM-метки, чтобы следить за результатами рекламы, продаж и любого контента на сайте.  
UTM расшифровывается как Urchin Tracking Module

***Позитивные vs Негативные тест-кейсы***

Тест кейсы разделяются по ожидаемому результату на **позитивные** и **негативные**:

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

***SDL vs SDLC***

System Life Cycle is basically the same thing as SDLC, with the added step of Disposal/End of life actions. S Development LC does not concern itself with getting rid of the items once they've reached the end of their usefulness.

***CRUD*** — [акроним](https://ru.wikipedia.org/wiki/%D0%90%D0%BA%D1%80%D0%BE%D0%BD%D0%B8%D0%BC), обозначающий четыре базовые функции, используемые при работе с [базами данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)[[1]](https://ru.wikipedia.org/wiki/CRUD#cite_note-1): создание ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *create*), чтение (*read*), модификация (*update*), удаление (*delete*). Введён [Джеймсом Мартином](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%B0%D1%80%D1%82%D0%B8%D0%BD,_%D0%94%D0%B6%D0%B5%D0%B9%D0%BC%D1%81_(%D1%83%D1%87%D1%91%D0%BD%D1%8B%D0%B9)&action=edit&redlink=1) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) [*James Martin*](https://en.wikipedia.org/wiki/James_Martin_(author))) в 1983 году[[2]](https://ru.wikipedia.org/wiki/CRUD#cite_note-2) как стандартная классификация функций по манипуляции данными.

В SQL этим функциям, операциям соответствуют операторы [Insert](https://ru.wikipedia.org/wiki/Insert_(SQL)" \o "Insert (SQL)) (создание записей), [Select](https://ru.wikipedia.org/wiki/Select_(SQL)" \o "Select (SQL)) (чтение записей), [Update](https://ru.wikipedia.org/wiki/Update_(SQL)" \o "Update (SQL)) (редактирование записей), [Delete](https://ru.wikipedia.org/wiki/Delete_(SQL)" \o "Delete (SQL)) (удаление записей). В некоторых CASE-средствах использовались специализированные CRUD-матрицы или CRUD-диаграммы, в которых для каждой сущности указывалось, какие базовые функции с этой сущностью выполняет тот или иной процесс или та или иная роль. В системах, реализующих доступ к базе данных через API в стиле [REST](https://ru.wikipedia.org/wiki/REST), эти функции реализуются зачастую (но не обязательно) через HTTP-методы PUT, GET, PATCH и DELETE соответственно.

**Unix vs linux**

Чем похожи Linux и Unix?

Отличий в этих системах намного больше чем похожего. Но все же следует обратить внимание на то, что обе Linux и Unix системы имеют похожие принципы администрирования данных, кроме того базовые команды управления ОС идентичны. Также системы устойчивы к вирусам на одном уровне. Если вы работали уже с одной системой, то со второй будет разобраться намного проще, если бы вы начали обучаться системе с нуля.

Названия **Linux и Unix очень похожи, но это не означает,  что в этих системах много общего.** Перейдем сразу к основным отличиям.

Какие основные отличия между Linux и Unix?

Linux - ОС с открытым исходным кодом, распространяется бесплатно, Unix - только ее производные находятся в свободном доступе. Linux часто называют своеобразным клоном Unix, который не использует его код. Linux изначально разрабатывался для домашних ПК, а Unix для больших корпораций и тех, кто занимается программной разработкой. Правда, на сегодня Linux поддерживает больше платформ чем Unix и является более популярным среди пользователей. И, конечно же, Linux поддерживает больше типов файловых систем чем Unix.

Часто данные понятия путают из-за того, что они могут по-разному трактоваться и обозначать разные вещи. Если учитывать все факторы, то ОС Unix менее универсальна, потому что каждый дистрибутив адаптирован к только определенному количеству платформ и файловых систем, в отличии от Linux, которая в данном плане намного универсальней и практичней.

О Linux знают намного больше пользователей чем о UNIX, но UNIX также незаменима в свой среде использования.

Отличия между Linux и UNIX значительны. **UNIX - широкое понятие**, конкретнее говоря, некий фундамент для построения и сертификации всех UNIX-подобных систем, у unix преимущества и недостатки есть также, как и у остальных ОС; а Linux - одна из веток, UNIX-подобная, но полностью самостоятельная система. Хотя, если учитывать фактор технологий, то данные термины довольно близки, но очень разные в то же время.

Много ПО, которое было создано для Linux, можно отлично адаптировать для работы с ОС UNIX. При этом не нужно переписывать исходный код. Если вы пользователь линукс, тогда вам обязательно пригодится следующая статья: Полезные команды Linux, которые необходимо знать в следующей[статье](https://hyperhost.ua/info/poleznyie-komandyi-linux-kotoryie-neobhodim/).

А теперь подытожим все основные отличия между **Linux и Unix в сравнительной таблице**:

|  |  |  |
| --- | --- | --- |
| **Критерий сравнения** | **Unix** | **Linux** |
| Разработка | AT & T Bell Labs в США | энтузиасты, изначально Линус Торвальдс |
| Владелец ТМ | Open Group | Linux Mark Institute при Linux Foundation |
| Выпуск 1-го релиза | 1973 год | 1991 год |
| Дистрибутивы | IBM AIX, Solaris, HP-UX, Darwin, macOS и др. | Ubuntu, Debian, CentOS,  Fedora, Manjaro, Arch  и др. |
| GUI | Gnome | Gnome, KDE и Xfce |
| Исходный код | Закрытый и открытый в некоторых системах | Открытый |
| Язык программирования | Написан на Си/Ассемблер | Написан на Си и др. |
| Поддержка | Платная | Бесплатная |
| Интерфейс | sh и другие совместимые | BASH |
| Ядро | Разные виды ядра | Монолитное |
| Для чего используют | Для серверов, где есть ПО с особыми требованиями | Для ПК, серверов, ЦОД, облачных систем |
| Цена | Стоимость устанавливается поставщиками | Бесплатно. Есть некоторые платные дешевые дистрибутивы |

Вывод статьи

Как итог данной статьи можно привести вот такую последовательность **ОС Юникс → UNIX-подобная ОС → Linux**, которая во многом объясняет зависимость обеих операционных систем.  Оба понятия Юникс и Линукс в зависимости от контекста могут обозначать абсолютно разные значения. Unix появился первым, затем появился Linux. Именно Линукс был вдохновлен работой Unix, но стал более свободным, независимым ПО, которое значительно отличается от Unix и имеет свои цели использования.

## Что такое URL адрес: структура и значение для SEO [инфографика]Тест дизайн? (Test Design)

Этап процесса тестирования ПО, на котором проектируются и создаются Test case (тест кейсы), в соответствии с определенными ранее критериями качества и целями тестирования.

План работы над тест дизайном:

* анализ имеющихся проектных артефактов: документация (спецификации, требования, планы), модели, исполняемый код и т. д.
* написание спецификации по тест дизайну ([Test Design Specification](https://www.google.com/url?q=http://www.protesting.ru/documentation/sqe_test_design_specification_template.zip&sa=D&source=editors&ust=1613993129160000&usg=AOvVaw0dZ52-fa2-wpMirsg3iCGi))
* проектирование и создание [Test case](https://www.google.com/url?q=http://www.protesting.ru/testing/testcase.html&sa=D&source=editors&ust=1613993129161000&usg=AOvVaw25Cnm9iNncXSqLm-XBLxtd)

**Тонкий vs толстый клиент**

Толстые клиенты работают с информацией на основе собственных аппаратных и программных возможностей, в то же время тонкие применяют ПО центрального сервера только чтобы обработать данные, предоставляя системе лишь требуемый графический интерфейс для выполнения работы пользователем. Это значит, что в роли тонких клиентов иногда мы можем увидеть устаревшие или не очень производительные ПК.

Вася хочет перейти на сайт FirstSSL, защищённый SSL-сертификатом

→ Васин браузер посылает запрос к сайту

→ сайт отправляет в ответ копию сертификата

→ браузер проверяет подлинность сертификата — узнаёт у центра сертификации, который его выдал

→ если сертификат не поддельный, сайт и браузер тайно [договариваются о секретном симметричном ключе](https://www.firstssl.ru/faq/general-questions/kluch-shifrovania?_ga=2.215362455.108052350.1631629895-1906762590.1631629895)

## В чем отличия тестирования мобильного приложения от десктопного?

* В первую очередь это разные операционные системы и разная архитектура «железа», хотя сейчас прогресс нацелен на унификацию (например, такие гиганты, как Microsoft и Apple. У MS это планшеты-ноутбуки Surface на базе ARM и Windows 10, Apple в июне 2020 года заявила о переходе на ARM-архитектуру в компьютерах).
* Пока еще актуальное различие – аппаратные ресурсы. Мощность начинки и количество памяти.
* Самое очевидное различие в аппаратной части помимо мощности – наличие разнообразных датчиков и модулей связи в мобильном устройстве, а также нескольких камер, вибромотора, сканера отпечатков и т. д.
* Наличие датчика ориентации уже предполагает тестирование в портретной и ландшафтной ориентациях. Добавьте к этому множество разрешений дисплея, их различные типы матриц со своими особенностями отображения и т.п.
* Помимо этого, в мобильном устройстве очень большое внимание уделяется обработке разнообразных прерываний (входящий звонок, уведомление, нажатие кнопки блокировки, выгрузка из ОЗУ и т. д. )
* Основная функция мобильных устройств – по-прежнему связь. Голосовая, но также и через мобильный интернет, что усложняет тестирование по сравнению с десктопами.
* Прогресс в обновлениях ОС. В мобильных устройствах это происходит гораздо чаще и имеет большее значение в связи с большой конкуренцией.
* Различия в гайдлайнах для ОС.
* Десктопные приложения чаще всего загружаются с официального веб-сайта производителя. Мобильное приложения почти всегда загружается из соответствующего ОС магазина приложений.

## В чем отличия тестирования мобильного приложения от web?

* Большее количество вариантов и комбинаций ОС/железа и т.п. в мобильных устройствах (сюда же вытекающее следствие необходимости использования эмуляторов)
* Браузеры «стационарны», в то время как при тестировании мобильных приложений нужно учитывать ориентацию, прерывания, связь, наличие дополнительных модулей.
* С т.з. связи веб приложение фактически становится бесполезным при потере интернет-соединения (хотя в последнее время это иногда не совсем так), для нативного мобильного приложения ничего не изменится\*.
* Публикация и распространение. Для того, чтобы люди начали пользоваться мобильным приложением, необходим аккаунт разработчика и пройденная модерация в магазине приложений.