**Клиент - сервер** - 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, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных);
* сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине).

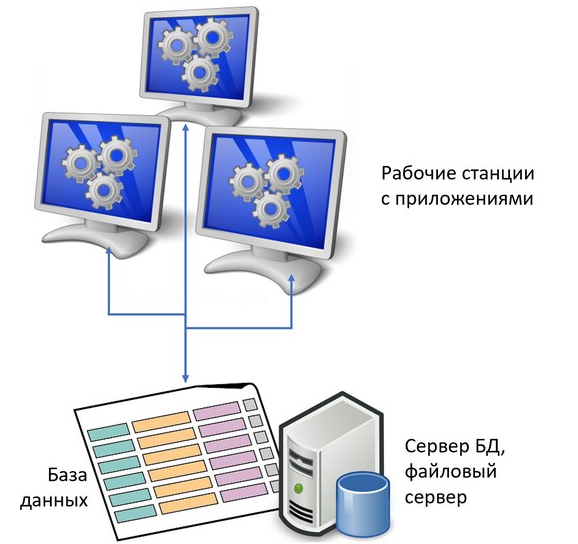
Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, её также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Архитектуру «клиент-сервер» принято разделять на три класса: одно-, двух- и трехуровневую. Однако, нельзя сказать, что в вопросе о таком разделении в сообществе ИТ-специалистов существует полный консенсус. Многие называют одноуровневую архитектуру двухуровневой и наоборот, то же можно сказать о соотношении двух- и трёхуровневой архитектур.

**Одноуровневая архитектура (1-Tier)**

Одноуровневая архитектура «клиент-сервер» (1-Tier) - такая, где все прикладные программы рассредоточены по рабочим станциям, которые обращаются к общему серверу баз данных или к общему файловому серверу. Никаких прикладных программ сервер при этом не исполняет, только предоставляет данные.

В целом, такая архитектура очень надежна, однако, ей сложно управлять, поскольку в каждой рабочей станции данные будут присутствовать в разных вариантах. Поэтому возникает проблема их синхронизации на отдельных машинах. В общем, как можно видеть из рисунка, в этой архитектуре просматривается еще один уровень - базы данных, что дает повод во многих случаях называть её двухуровневой.



**Двухуровневая архитектура (2-Tier)**

К двухуровневой архитектуре «клиент-сервер» следует относить такую, в которой прикладные программы сосредоточены на сервере приложений (Application Server), например, сервере 1С или сервере CRM, а в рабочих станциях находятся программы-клиенты, которые предоставляют для пользователей интерфейс для работы с приложениями на общем сервере.

Такая архитектура представляется наиболее логичной для архитектуры «клиент-сервер». В ней, однако, можно выделить два варианта. Когда общие данные хранятся на сервере, а логика их обработки и бизнес-данные хранятся на клиентской машине, то такая архитектура носит название “**fat client thin server**” (толстый клиент, тонкий сервер). Когда не только данные, но и логика их обработки и бизнес-данные хранятся на сервере, то это называется “**thin client fat server**” (тонкий клиент, толстый сервер). Такая архитектура послужила прообразом облачных вычислений (Cloud Computing).



Преимущества двухуровневой архитектуры:

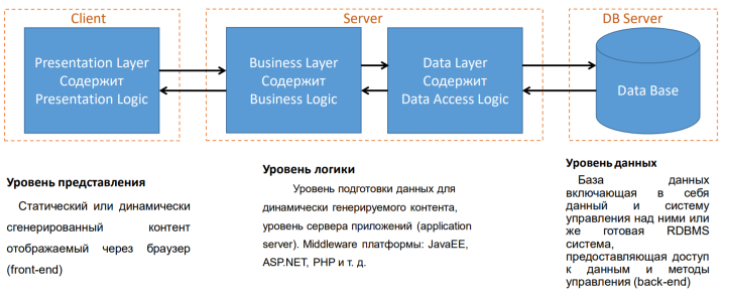
* Легко конфигурировать и модифицировать приложения;
* Пользователю обычно легко работать в такой среде;
* Хорошая производительность и масштабируемость.

Однако, у двухуровневой архитектуры есть и ограничения:

* Производительность может падать при увеличении числа пользователей;
* Потенциальные проблемы с безопасностью, поскольку все данные и программы находятся на центральном сервере;
* Все клиенты зависимы от базы данных одного производителя;

**Трехуровневая архитектура (3-Tier)**

В трехуровневой архитектуре сервер баз данных, файловый сервер и другие представляют собой отдельный уровень, результаты работы которого использует сервер приложений. Логика данных и бизнес-логика находятся в сервере приложений. Все обращения клиентов к базе данных происходят через промежуточное программное обеспечение (middleware), которое находится на сервере приложений. Вследствие этого, повышается гибкость работы и производительность.



Преимущества трёхуровневой архитектуры:

* Целостность данных;
* Более высокая безопасность, по сравнению с двухуровневой архитектурой;
* Защищённость базы данных от несанкционированного проникновения.

Ограничения:

* Более сложная структура коммуникаций между клиентов и сервером, поскольку в нём также находится middleware.

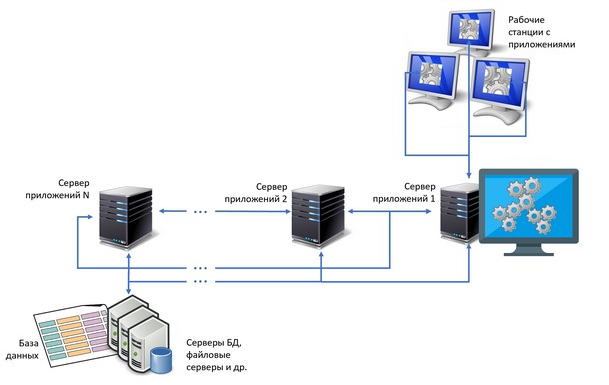
**Многоуровневая архитектура (N-Tier)**

В отдельный класс архитектуры «клиент-сервер» можно вынести многоуровневую архитектуру, в которой несколько серверов приложений используют результаты работы друг друга, а также данные от различных серверов баз данных, файловых серверов и других видов серверов.

По сути, предыдущий вариант, трехуровневая архитектура - не более, чем частный случай многоуровневой архитектуры.

Преимуществом многоуровневой архитектуры является гибкость предоставления услуг, которые могут являться комбинацией работы различных приложений серверов разных уровней и элементов этих приложений.

Очевидным недостатком является сложность, многокомпонентность такой архитектуры.



**Характеристики архитектуры «клиент-сервер»**

* **Асимметричность протоколов**. Между клиентами и сервером существуют отношения «один ко многим». Инициатором диалога с сервером обычно является клиент.
* **Инкапсуляция услуг**. После получения запроса на услугу от клиента, сервер решает, как должна быть выполнена данная услуга. Модификация («апгрейд») сервера может производиться без влияния на работу клиентов, поскольку это не влияет на опубликованный интерфейс взаимодействия между ними. Иными словами, максимум, что может при этом почувствовать пользователь - незначительная задержка отклика сервера в течение небольшого времени апгрейда.
* **Целостность**. Программы и общие данные для сервера управляются централизованно, что снижает стоимость обслуживания и защищает целостность данных. В то же время, данные клиентов остаются персонифицированными и независимыми.
* **Местная прозрачность**. Сервер - это программный процесс, который может исполняться на той же машине, что и клиент, либо на другой машине, подключенной по сети. Программное обеспечение «клиент-сервер» обычно скрывает местоположение сервера от клиентов, перенаправляя запрос на услуги через сеть.
* **Обмен на основе сообщений**. Клиенты и сервер являются нежёстко связанными («loosely-coupled») процессами, которые обмениваются сообщениями: запросами на услуги и ответами на них.
* **Модульный дизайн**, способный к расширению. Модульный дизайн программной платформы «клиент-сервер» придаёт ей устойчивость к отказам, то есть, отказ в каком-то модуле не вызывает отказа всего приложения. В такой системе, один или больше серверов могут отказать без остановки всей системы в целом, до тех пор, пока услуги отказавшего сервера могут быть предоставлены с резервного сервера. Другое преимущество модульности в том, что приложение «клиент-сервер» может автоматически реагировать на повышение или понижение нагрузки на систему, путем добавления или отключения услуг или серверов.
* **Независимость от платформы**. Идеальное приложение «клиент-сервер» не зависит от платформ оборудования или операционной системы. Клиенты и серверы могут развертываться на различных аппаратных платформах и разных операционных системах.
* **Масштабируемость**. Системы «клиент-сервер» могут масштабироваться как горизонтально (по числу серверов и клиентов), так и вертикально (по производительности и спектру услуг).
* **Разделение функционала**. Система «клиент-сервер» - это соотношение между процессами, работающими на одной или на разных машинах. Сервер - это процесс предоставления услуг. Клиент - это потребитель услуг.
* **Общее использование ресурсов**. Один сервер может предоставлять услуги множеству клиентов одновременно, и регулировать их доступ к совместно используемым ресурсам.

**Практические применения архитектуры «клиент-сервер»**

Архитектура «клиент-сервер» - один из основных принципов работы сети Интернет. Любой веб-сайт, или приложение в Интернет работает на сервере, а его пользователи являются клиентами. Социальные сети (Фейсбук, ВК и пр.), сайты электронной коммерции (Amazon, Озон и др.) , мобильные приложения (Instagram и т.д.), устройства Интернета вещей (умные колонки или смарт-часы) работают на основе клиент-серверной архитектуры.

Хорошим примером работы системы «клиент-сервер» является автомобильный навигатор. Приложение навигации на сервере собирает данные с многих смартфонов пользователей, на которых установлены клиенты приложения. Кроме того, приложение навигации использует ещё и данные с сервера базы данных - геоинформационной системы, который предоставляет данные, например, о текущих ремонтах дорог, о появлении новых дорог и пр. Данные со многих клиентов (местоположение, скорость) обрабатывается сервером навигации и выдаётся на смартфоны пользователей в виде информации о средней скорости движения по тому или иному участку маршрута.

Практически любая корпоративная сеть или ИТ-система предприятия, как правило, строится по архитектуре «клиент-сервер». В небольших сетях (3-5 компьютеров в компании) функции сервера может выполнять один из рабочих компьютеров. Если число машин в организации более 10, то лучше сделать выделенный сервер (почтовый сервер, приложений, баз данных и пр.), который будет заниматься обслуживанием клиентов - компьютеров и телефонов сотрудников организации.

В домашних сетях архитектура «клиент-сервер» тоже используется довольно часто. Например, в домашнюю сеть могут быть объединены компьютеры членов семьи, один из которых выполняет функции сервера. В домашнюю сеть также могут быть включены такие устройства, как умные колонки, умные домашние устройства (пылесосы-роботы, фотоаппараты, DVD-плееры и пр.), а также «умные» счётчики (вода, электричество) и т.д. Тогда в системе управления сервера, будут видны все параметры, данные и медиа файлы (музыка, видео, фото), а также «умные устройства».

В настоящее время можно встретить термин Serverless Architecture, т.н. «бессерверная архитектура». Однако, по сути, она представляет собой процесс получения функций сервера в виде облачной услуги. То есть, серверы в облаке тоже есть, но для конечного пользователя они не видны, и он получает их сервисы в виде абстрактной «функции как услуги» FaaS (Function as a Service).

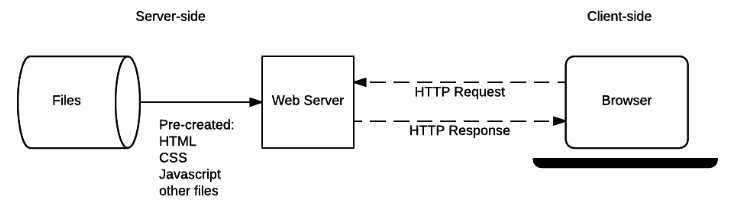
Архитектура «клиент-сервер» является основой большинства корпоративных сетей и берет свое начало от самых первых вычислительных машин, т.н. «мэйнфреймов».

*Примечание. Можно встретить такие понятия:*

* *“Толстый” клиент: на сервере реализованы главным образом функции доступа к базам данных, а основные прикладные вычисления выполняются на стороне клиента;*
* *“Тонкий клиент”: на сервере выполняется основная часть прикладной обработки данных, а на клиентские рабочие станции передаются уже результаты обработки данных для просмотра и анализа пользователем с возможностью их последующей обработки (в минимальном объеме).*
* *“Горячий/теплый” клиент: с кэшем и куками;*
* *“Холодный” клиент: чистый, без или очищеный от кук и кэша.*

**Статический сайт** - это тот, который возвращает тот же жёсткий кодированный контент с сервера всякий раз, когда запрашивается конкретный ресурс. Например, если у вас есть страница о товаре в /static/myproduct1.html, эта же страница будет возвращена каждому пользователю. Если вы добавите еще один подобный товар на свой сайт, вам нужно будет добавить ещё одну страницу (например, myproduct2.html) и так далее. Это может стать действительно неэффективным - что происходит, когда вы попадаете на тысячи страниц товаров? Вы повторяли бы много кода на каждой странице (основной шаблон страницы, структуру и т. д.), И если бы вы захотели изменить что-либо в структуре страницы - например, добавить новый раздел «связанные товары» - тогда вам придётся менять каждую страницу отдельно.

На заметку: Статические сайты превосходны, когда у вас небольшое количество страниц и вы хотите отправить один и тот же контент каждому пользователю. Однако их обслуживание может потребовать значительных затрат по мере увеличения количества страниц.



Когда пользователь хочет перейти на страницу, браузер отправляет HTTP-запрос GET с указанием URL-адреса его HTML-страницы. Сервер извлекает запрошенный документ из своей файловой системы и возвращает HTTP-ответ, содержащий документ и код состояния HTTP Response status code 200 OK (успех). Сервер может вернуть другой код состояния, например, «404 Not Found», если файл отсутствует на сервере или «301 Moved Permanently», если файл существует, но был перемещён в другое место.

Серверу для статического сайта нужно будет только обрабатывать GET-запросы, потому что сервер не сохраняет никаких модифицируемых данных. Он также не изменяет свои ответы на основе данных HTTP-запроса (например, URL-параметров или файлов cookie).

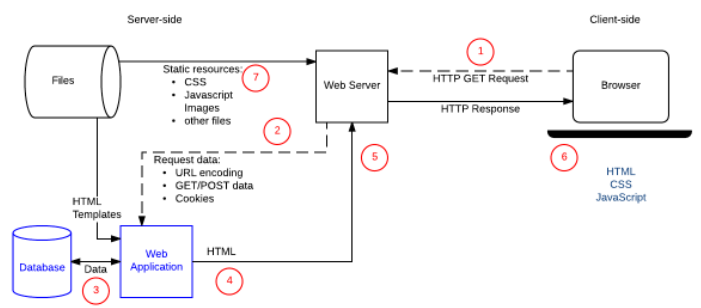
Понимание того, как работают статические сайты, тем не менее полезно при изучении программирования на стороне сервера, поскольку динамические сайты точно так же обрабатывают запросы для статических файлов (CSS, JavaScript, статические изображения и т. д.).

**Динамический сайт** - это тот, который может генерировать и возвращать контент на основе конкретного URL-адреса запроса и данных (а не всегда возвращать один и тот же жёсткий код для определенного URL-адреса). Используя пример сайта товара, сервер будет хранить «данные» товара в базе данных, а не отдельные HTML-файлы. При получении GET-запроса для товара сервер определяет идентификатор товара, извлекает данные из базы данных и затем создает HTML-страницу для ответа, вставляя данные в HTML-шаблон. Это имеет большие преимущества перед статическим сайтом:

* Использование базы данных позволяет эффективно хранить информацию о товаре с помощью легко расширяемого, изменяемого и доступного для поиска способа.
* Использование HTML-шаблонов позволяет очень легко изменить структуру HTML, потому что это нужно делать только в одном месте, в одном шаблоне, а не через потенциально тысячи статических страниц.

**Анатомия динамического запроса**: чтобы не отдаляться от практики, мы будем использовать контекст веб-сайта менеджера спортивной команды, где тренер может выбрать имя своей команды и размер команды в HTML-форме и вернуться к предлагаемому «лучшему составу» для своей следующей игры.

На приведённой ниже диаграмме показаны основные элементы веб-сайта «team coach», а также пронумерованные ярлыки для последовательности операций, когда тренер обращается к списку «лучших команд». Частями сайта, которые делают его динамичным, являются веб-приложение (так мы будем ссылаться на серверный код, обрабатывающий HTTP-запросы и возвращающие HTTP-ответы), база данных, которая содержит информацию об игроках, командах, тренерах и их отношениях, и HTML-шаблоны



После того, как тренер отправит форму с именем команды и количеством игроков, последовательность операций будет следующей:

1. Веб-браузер отправит HTTP-запрос GET на сервер с использованием базового URL-адреса ресурса (/best) и кодирования номера команды и игрока в форме URL-параметров (например, /best?team=my\_team\_name&show=11) или как часть URL-адреса (например, /best/my\_team\_name/11/). Запрос GET используется, потому что речь идёт только о запросе выборки данных (а не об их изменении).
2. Веб-сервер определяет, что запрос является «динамическим» и пересылает его в веб-приложение для обработки (веб-сервер определяет, как обрабатывать разные URL-адреса на основе правил сопоставления шаблонов, определённых в его конфигурации).
3. Веб-приложение определяет, что цель запроса состоит в том, чтобы получить «лучший список команд» на основе URL (/best/) и узнать имя команды и количество игроков из URL-адреса. Затем веб-приложение получает требуемую информацию из базы данных (используя дополнительные «внутренние» параметры, чтобы определить, какие игроки являются «лучшими», и, возможно, определяя личность зарегистрированного тренера из файла cookie на стороне клиента).
4. Веб-приложение динамически создаёт HTML-страницу, помещая данные (из базы данных) в заполнители внутри HTML-шаблона.
5. Веб-приложение возвращает сгенерированный HTML в веб-браузер (через веб-сервер) вместе с кодом состояния HTTP 200 («успех»). Если что-либо препятствует возврату HTML, веб-приложение вернёт другой код, например, «404», чтобы указать, что команда не существует.
6. Затем веб-браузер начнёт обрабатывать возвращённый HTML, отправив отдельные запросы, чтобы получить любые другие файлы CSS или JavaScript, на которые он ссылается (см. шаг 7).
7. Веб-сервер загружает статические файлы из файловой системы и возвращает их непосредственно в браузер (опять же, правильная обработка файлов основана на правилах конфигурации и сопоставлении шаблонов URL).

Операция по обновлению записи в базе данных будет обрабатываться аналогичным образом, за исключением того, что, как и любое обновление базы данных, HTTP-запрос из браузера должен быть закодирован как запрос POST.

**Выполнение другой работы**: задача веб-приложения - получать HTTP-запросы и возвращать HTTP-ответы. Хотя взаимодействие с базой данных для получения или обновления информации является очень распространённой задачей, код может делать другие вещи одновременно или вообще не взаимодействовать с базой данных.

Хорошим примером дополнительной задачи, которую может выполнять веб-приложение, является отправка электронной почты пользователям для подтверждения их регистрации на сайте. Сайт также может выполнять протоколирование или другие операции.

**Возвращение чего-то другого, кроме HTML**: серверный код сайта может возвращать не только HTML-фрагменты и файлы в ответе. Он может динамически создавать и возвращать другие типы файлов (текст, PDF, CSV и т. д.) или даже данные (JSON, XML и т. д.).

Идея вернуть данные в веб-браузер, чтобы он мог динамически обновлять свой собственный контент (AJAX) существует довольно давно. Совсем недавно «Одностраничные приложения» стали популярными, где весь сайт написан с одним HTML-файлом, который динамически обновляется по мере необходимости. Веб-сайты, созданные с использованием приложений такого рода, переносят большие вычислительные затраты с сервера на веб-браузер и приводят к тому, что веб-сайты, ведут себя больше как нативные приложения (очень отзывчивые и т. д.).