23.1 Что происходит, когда пользователь набирает в браузере адрес сайта?

За работу любого сайта обычно отвечает один из миллионов серверов, подключенных к интернету. Адрес сервера — это уникальный набор цифр, который называется IP-адресом. Например это сервер 85.119.149.83.

Поэтому первым делом браузеру нужно понять, какой IP-адрес у сервера, на котором находится сайт.

Такая информация хранится в распределенной системе серверов — **DNS** (Domain Name System). Система работает как общая «контактная книга», хранящаяся на распределенных серверах и устройствах в интернете.

Однако перед тем, как обращаться к DNS, браузер пытается найти запись об IP-адресе сайта в ближайших местах, чтобы сэкономить время:

- Сначала в своей истории подключений. Если пользователь уже посещал сайт, то в браузере могла сохраниться информация с IP-адресом сервера.
- В операционной системе . Не обнаружив информации у себя, браузер обращается к операционной системе, которая также могла сохранить у себя DNS-запись. Например, если подключение с сайтом устанавливалось через одно из установленных на компьютере приложений.

- В кэше роутера, который сохраняет информацию о последних соединениях, совершенных из локальной сети.

Не обнаружив подходящих записей в кэше, браузер формирует запрос к DNS-серверам, расположенным в интернете.

Например, если нужно найти IP-адрес сайта mail.vc.ru, браузер спрашивает у ближайшего DNS-сервера «Какой IP-адрес у сайта main.myfreedom.by?».

Сервер может ответить: «Я не знаю про main.myfreedom.by, но знаю сервер, который отвечает за myfreedom.by». Запрос переадресовывается дальше, на сервер «выше», пока в итоге один из серверов не найдет ответ об IP-адресе для сайта.

Если на любом из этапов находится нужная запись, то она сохраняется во всех кэшах и управление возвращается браузеру, который уже знает IP нужного сервера.

> Процесс получения IP адреса называется DNS lookup.

Далее Browser Engine смотрит в локальном кэше, нет ли запрашиваемой страницы. Если страницы в кэше нет, то Browser Engine передаёт управление компоненту Rendering Engine, который обращается к компоненту Networking Component, чтобы тот сделал запрос GET на указанный IP на порт 80 по протоколу HTTP или на порт 443 по протоколу HTTPS (в зависимости от указанного протокола в URL) со своими стандартными HTTP заголовками.

Среди стандартных заголовков есть заголовок **host**, в котором передаётся хост запрашиваемого сайта (в нашем примере main.myfreedom.by). Если в браузере хранятся куки к этому домену, то он отправляет их в заголовке **cookie**. Запрос

будет выполнен, если соответствующий порт на пользовательском компьютере открыт.

Это сообщение и все остальные данные, передаваемые между клиентом и сервером, передаются по интернет-соединению с использованием протокола TCP/IP.

В общении браузера и сервера выделяют два типа запросов. GET-запрос используется для получения данных с сервера — например, отобразить картинку, текст или видео. POST-запрос — используется для отправки данных из браузера на сервер, например, когда пользователь отправляет сообщение, картинку или загружает файл.

Почти все сайты обмениваются информацией с сервером в зашифрованном формате — с помощью HTTPS-протокола. В отличие от HTTP-протокола, в HTTPS используется шифрование, а безопасность подключения подтверждается специальным сертификатом.

На сервере запрос принимает веб-сервер (например, nginx или apache).

В конфигурационных файлах веб-сервера прописаны обслуживаемые хосты. Веб-сервер достаёт хост из заголовка запроса host и сопоставляет с теми, которые указаны в конфигурации. Если есть совпадение, то веб-сервер находит в конфигурационном файле правила обработки такого запроса и выполняет их. Дальнейшее поведение сервера зависит от технологии и особенностей приложения. Здесь может происходить работа с базами данных, кэшами, запросы к другим серверам и сервисам, выполнение различных скриптов. Для простоты представим, что приложение сгенерировало файл HTML, и веб-сервер отдал его браузеру.

Браузер получил файл HTML с соответствующими HTTP заголовками от сервера, в которых указана длина контента (заголовок Content-Length), тип контента

(заголовок Content-Type со значением “text/html; charset=UTF-8” для файлов HTML), заголовки для кэширования. Если присутствуют кэширующие заголовки, то браузер сохраняет файл в локальный кэш.

> Заголовки ответа сервера можно увидеть в Chrome DevTools на вкладке Networking, выбрав нужный запрос

Если длина контента больше нуля и тип контента поддерживается браузером, то браузер пытается его обработать. В нашем случае браузер получает файл HTML с соответствующим заголовком Content-Type. Браузер начинает разбор (parsing) этого файла с первой инструкции, которой является инструкция <!DOCTYPE>. DOCTYPE указывает на версию HTML, чтобы браузер понимал, каким правилам следовать во время разбора (какие теги как обрабатывать).

Если DOCTYPE отсутствует, то браузер переключится в режим quirks mode и попытается разобрать документ HTML, однако многие элементы будут проигнорированы. Если указан корректный DOCTYPE, то браузер будет работать в standards mode и будет разбирать документ в соответствии с правилами той версии, которая указана в DOCTYPE.

Rendering Engine начинает разбор документа HTML.

Создаётся DOM (Document Object Model). В браузере этот объект доступен по ссылке, которая хранится в переменной **document**. У документа есть несколько состояний. Первое состояние — loading. Оно означает, что документ только начал формироваться.

Создаётся DOM (Document Object Model). В браузере этот объект доступен по ссылке, которая хранится в переменной **document**. У документа есть несколько

состояний. Первое состояние — loading. Оно означает, что документ только начал формироваться.

> Состояние документа хранится в переменной document.readyState.

Также создаётся объект styleSheets, который будет хранить все стили.

> Все стили на странице доступны по ссылке, которая хранится в переменной document.styleSheets.

Любой файл — это набор байтов. Браузер берёт полученный набор байтов и преобразует их в символы по таблице символов в соответствии с кодировкой, которая была передана в заголовке **Content-Type**. В нашем примере это кодировка UTF-8.

Следующий процесс —разбивание текста на смысловые блоки (tokenization). Так браузер распознаёт теги <html>, <head> и проч., а также понимает, какие правила к какому тегу применять (например, поддерживаемые атрибуты).

Далее токены собираются в узлы (nodes). Эти узлы и сохраняются в DOM со всеми взаимными связями.

Во время разбора, если Rendering Engine встречает ссылку на внешний ресурс, то он передаёт команду загрузить этот ресурс компоненту Networking Component. Это может быть ссылка на стили, скрипты, картинки и т.п. Networking Component ставит все ресурсы в очередь на загрузку. Каждому ресурсу Networking Component присваивает приоритет.

> Приоритеты ресурсов можно посмотреть в Chrome DevTools на вкладке Networking в колонке Priority.

Так, у HTML, CSS и шрифтов самый высокий приоритет. У изображений приоритет изначально низкий, но если Rendering Engine обнаружит, что изображение попадает в поле видимости (view port) пользователя, то повысит приоритет до среднего. Приоритет скрипта зависит от положения на странице и способа загрузки. У асинхронных скриптов (async/defer) низкий приоритет. У скриптов, которые в документе перед изображениями — высокий, у тех, что после хотя бы одного изображение — средний.

По возможности браузер пытается загружать ресурсы параллельно. Однако, он не может загружать параллельно более 6 ресурсов с одного домена.

Кроме того, когда Rendering Engine отдаёт команду компоненту Networking Component на синхронную загрузку стиля или скрипта, он останавливает разбор документа.

С загрузкой стилей происходит подобный процесс преобразования из байтов в Object Model (CSSOM): байты -> символы -> токены -> узлы -> CSSOM.

Немного иначе происходит загрузка скрипта. Вместо того, чтобы вернуть управление Rendering Engine’у, Networking Component . передаёт управление **JavaScript Interpreter**, который преобразует байты в исполняемый код: байты -> символы -> токены -> Abstract Syntax Tree (evaluating). Далее в работу вступает компилятор, который оптимизирует AST, кэширует некоторые участки кода, компилирует его на лету (JIT compilation) в исполняемый код и исполняет (executing). Однако исполняется скрипт только, когда готова CSSOM. До тех пор скрипт стоит в очереди на исполнение.

Во многих современных браузерах во время исполнения JavaScript в отдельном потоке продолжается сканирование документа на наличие ссылок на другие ресурсы и постановка ресурсов в очередь на скачивание (Speculative parsing).

> Каждый этап разбора HTML, CSS и JS можно увидеть в Chrome DevTools во вкладке Performance

Если при загрузке скрипта Rendering Engine видит у скрипта атрибут async, то он не останавливает разбор документа во время загрузки скрипта. Скрипт также станет в очередь на исполнение, дожидаясь, когда CSSOM будет готова.

Если при загрузке скрипта Rendering Engine видит у скрипта атрибут defer, то он не останавливает разбор документа во время загрузки скрипта, но когда скрипт загрузится, он станет в очередь на исполнение, которая заработает при возникновении события DOMContentLoaded. К этому моменту CSSOM будет уже готова.

Когда Rendering Engine заканчивает разбор документа, он вызывает событие **DOMContentLoaded**, и состояние документа меняется на interactive. При этом ресурсы (например, картинки) могут продолжать загружаться.

Когда все ресурсы загрузились, вызывается событие **load**, а состояние документа меняется на **complete**.

После того, как документ полностью разобран и сформированы DOM и CSSOM, Rendering Engine начинает построение **Render Tree**. В него попадут все элементы, которые нужно отрисовать. Некоторые элементы изначально могут быть невидимыми — их не нужно рисовать. Для каждого элемента, который “выпадает”

из потока (например, используется position: absolute), будет создаваться отдельная ветка в Render Tree.

Во время Rendering Tree происходит сопоставление узлов из DOM и узлов CSSOM.

> Свойства узла можно получить с помощью функции window.getComputedStyles(узел).

Когда Rendering Tree готов, Rendering Engine запускает процесс **layout**. Он заключается в вычислении размеров и позиций каждого элемента на странице.

Следующий этап — **paint**. Rendering Engine вычисляет цвет каждого пикселя.

И, наконец, последний этап — **composite**. Компонент **UI Backend** слой за слоем отрисовывает элементы на странице. При этом, если требуется отрисовать изображение, которое ещё не загрузилось, во время процесса layout, Rendering Engine зарезервирует место для изображения, если у него указаны ширина и высота. Rendering Engine вынесет на отдельный слой те элементы, стили которых содержат правила opacity, transform или will-change. Более того, эти слои Rendering Engine передаст для обработки GPU.

Если требуется отобразить текст, для которого используется нестандартный шрифт, то современные браузеры скроют текст до момента загрузки шрифта (flash of invisible text).

В современных браузерах скачивание документа, его разбор и отрисовка происходят по кускам, частями.

В документе HTML могут присутствовать некоторые мета-теги, которые могут менять порядок загрузки ресурсов, а также их приоритет.

К примеру, мета-тег **dns-prefetch** вынуждает Rendering Engine обратиться к Networking Component и получить IP нужного домена ещё до того, как Rendering Engine встретить его в документе.

Мета-тег **prefetch** вынудит Networking Component поставить указанный ресурс в очередь на загрузку с низким приоритетом.

Мета-тег **preload** вынудит Networking Component поставить указанный ресурс в очередь на загрузку с высоким приоритетом.

Мета-тег **preconnect** вынудит Networking Component заранее подключиться к другом хосту, то есть пройти нужные этапы: DNS lookup, redirects, hand shakes.

> Время, которое затрачивается на каждый этап, происходящий во время запроса и разбора документа HTML, хранится в объекте performance.timing.

Полезные ссылки

How Browsers Work: Behind the scenes of modern web browsers

How JavaScript works: Parsing, Abstract Syntax Trees (ASTs) + 5 tips on how to minimize parse time

Notes on “How Browsers Work”