Чеклист производительности Front-End для 2017

Кто из Вас уже использует прогрессивную загрузку (progressive booting) в своих проектах? А как на счет tree**-shaking и code-splitting** в *React* и *Angular*? Успели ли Вы настроить под себя сжатие Brotli или Zopfli, сшивания OCSP (OCSP stapling) и сжатие HPACK (HPACK compression)? Не стоит так же забывать о подсказках ресурсов (resource hints), клиентских подсказках (client hints) и сдерживании CSS— и это не говоря уже о IPv6, HTTP/2 и сервис воркерах?

Давайте вспомним дни, когда производительность часто откладывалась на второй план. Вспоминали о ней аж в конце проекта, и в большинстве случаев это сводилось к минификации, конкатенации и оптимизации содержимого, и, возможно, парой дополнительных строк в конфигурационном файле сервера. Оглядываясь на те времена, понимаешь, что ситуация значительно поменялась с тех пор.

Производительность касается не только технической стороны вопроса: она действительно важна, нужно учитывать, как может отразится на производительности то или иное решение в момент создания дизайна. Производительность нужно постоянно замерять, отслеживать, и улучшать, и вечно-растущая сложность веб-технологий все чаще бросает разработчику вызов, делая невозможным отслеживание метрик, так как метрики будут значительно отличаться в зависимости от устройства, браузера, сетевого протокола, типа сети и задержки в ней (CDN-ы, ISP, кэши, прокси, фаерволы, балансировщики нагрузки и сами сервера играют большую роль в конечной производительности).

Если бы Вам пришлось сделать обзор на все возможные технологии о которых придется помнить в случае если поставлена задача улучшить производительность Вашего сайта — от самого начала процесса и до финального релиза вебсайта — на что бы Вы обратили свое внимание? Ниже приведен (надеюсь, беспристрастный и объективный) **чеклист производительности** **front-end’а для 2017** — обзор всех тем с которыми вам возможно придется столкнуться для обеспечения максимально быстрого отклика и плавности работы Вашего вебсайта.

(Вы так же можете просто ***загрузить чеклист в PDF (ENG) (0.129 MB)*** / ***загрузить чеклист в PDF (RUS) (0.129 MB)*** или ***скачать чеклист с Apple Pages (ENG) (0.236 MB)****.* Cчастливой оптимизации!)

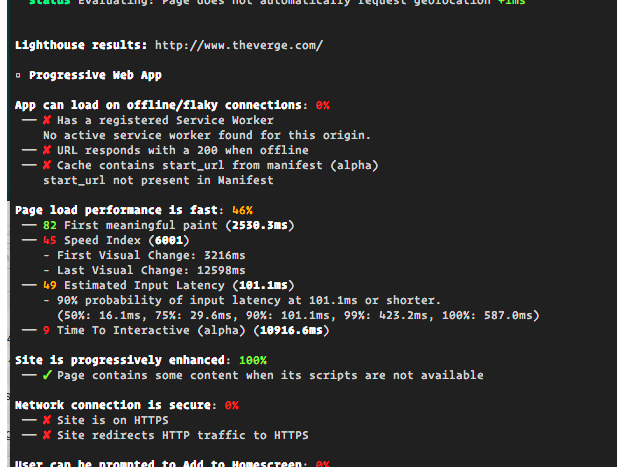
**Чеклист производительности Front-End’а в 2017**

Текущие микро-оптимизации — это отличная вещь, чтобы удерживать Ваш продукт в строю, но гораздо важнее предопределить какие цели перед Вами стоят и не забывать о них — измеримые цели, которые будут влиять на любые решения, принимаемые на протяжении всего процесса. Существует множество разнообразных моделей, и приведенные ниже – могут варьироваться, но главное — расставить в своем проекте правильные приоритеты.

**ГОТОВИМСЯ, СТАВИМ ЦЕЛИ**

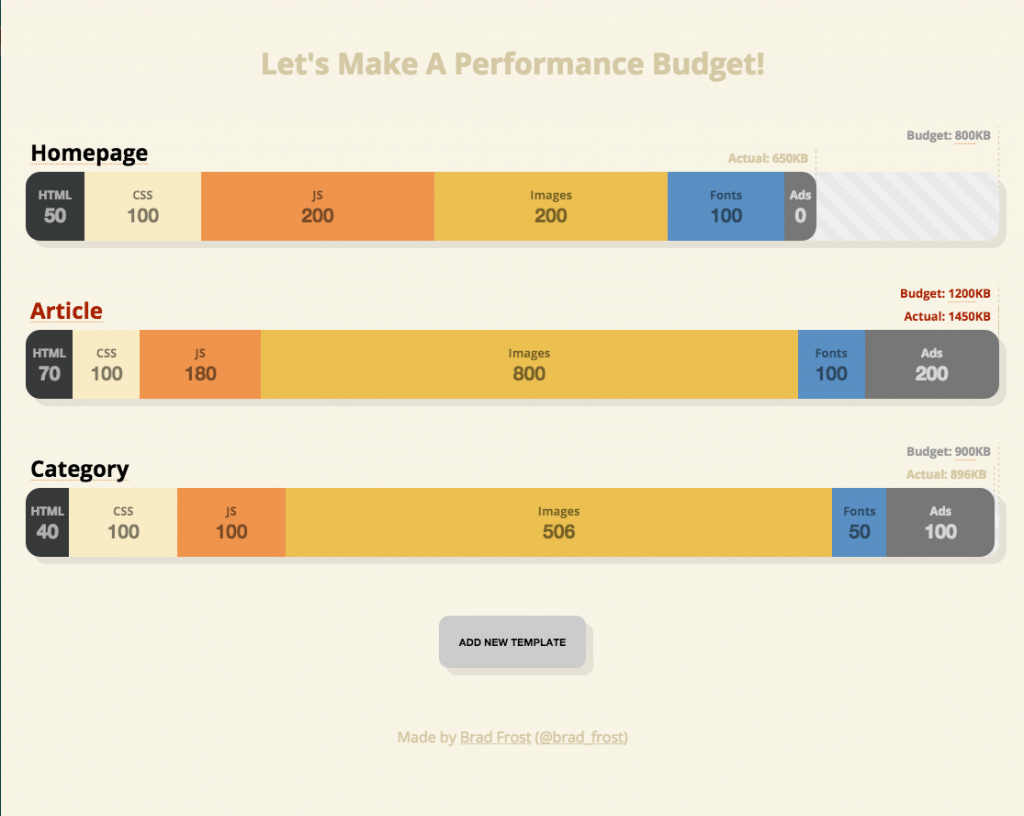
**1. Будь на 20% быстрее ближайшего соперника.**  
По результатам психологического исследования, оказалось что если вы хотите что пользователь ощущал что Ваш вебсайт быстрее других, нужно быть как-минимум на 20% быстрее. Не столь важно время полной загрузки и отображения страницы, как метрики, например, время начала рендера страницы, первое отображение значительной картинки ([first meaningful paint](https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint)) (к примеру, время необходимое что бы страница отобразила ее преймущественный контент) и время интерактивности ([time to interactive](https://developers.google.com/web/tools/lighthouse/audits/time-to-interactive)) (то есть, время необходимое странице или, в большинстве случаев, SPA что бы предоставить пользователю возможность взаимодействия с ней).

Измерить время начала рендера (с помощью [WebPagetest](http://www.webpagetest.org/)) и время отображения первой значимой картинки (с помощью [Lighthouse](https://github.com/GoogleChrome/lighthouse)) на Moto G, на устройстве Samsung среднего уровня производительности и на добротном среднестатистическом устройстве, например Nexus 4, желательно (*пометка переводчика* – для прогрессивного мира) в открытой лаборатории девайсов ([open device lab](https://www.smashingmagazine.com/2016/11/worlds-best-open-device-labs/)) — на стандартных скоростях 3G, 4G и при подключении к точке доступа Wi-Fi.

**

*Lighthouse, ПО для ревизии производительности от Google.*

Обратите внимание на Вашу аналитику, что бы определить, на чем зацикливаются пользователи. Тогда Вы сможете сымитировать 90% пользовательского экспириенса синтетическим тестированием. Соберите информацию, составьте таблицу, ужмите ее на 20% и таким образом определитесь с Вашими целями. (например, с бюджетом производительности ([performance budgets](http://bradfrost.com/blog/post/performance-budget-builder/)). Теперь у Вас есть что-то измеримое, на что нужно ориентироваться в тестировании. Держа в голове мысль о бюджете производительности, и предпринимая попытки написания хоть малейшего скрипта для уменьшения времени интерактивности Вы можете быть уверены, что вы на верном пути.



[Билдер](http://bradfrost.com/blog/post/performance-budget-builder/) бюджета производительности от Брэда Фроста.

**Поделитесь чеклистом с коллегами.** Удостоверьтесь что каждый член Вашей команды ознакомлен с данным чеклистом что бы избежать возможного недопонимания в будущем процессе. Каждое принятое решение имеет влияние на производительность, и проект будет только в выигрыше, если не только фронтендеры, но и UX и графические дизайнеры решаться следовать этому плану. Обозначьте решения дизайнера в бюджете производительности и расставьте приоритеты о которых упоминалось выше.

**2. Время отклика 100-миллисекунд, 60 кадров в секунду.**  
Модель производительности RAIL ([RAIL performance model](https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/)) ставит перед Вами амбициозные цели: приложите все усилия и предоставьте пользователю отклик в течении 100 миллисекунд после первого входа. Для создания возможности отклика <100 миллисекунд страница должна уступать управление главному потоку не позже чем через каждые <50 миллисекунд. Для точек повышенной нагрузки, как анимация, лучшим решением будет не добавлять ничего больше там, где Вы еще можете, и лишь самый минимум там, где вы уже не можете.

Более того, каждый кадр анимации должен сменятся менее чем за 16 миллисекунд, тем самым Вы достигнете заветных 60 fps (1 секунда ÷ 60 = 16.6 миллисекунд) — а еще лучше – уложиться в пределы 10 миллисекунд. Вашему браузеру ведь нужно время что бы отрисовать новый кадр на экране, так что постарайтесь что бы исполнение Вашего кода завершилось в течении 16.6 миллисекунд. Будьте оптимистом ( [Be optimistic](http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation) ) и используйте время простоя с умом. Очевидно, что эта цель относится в большей степени к производительности во время выполнения, а не во время загрузки.

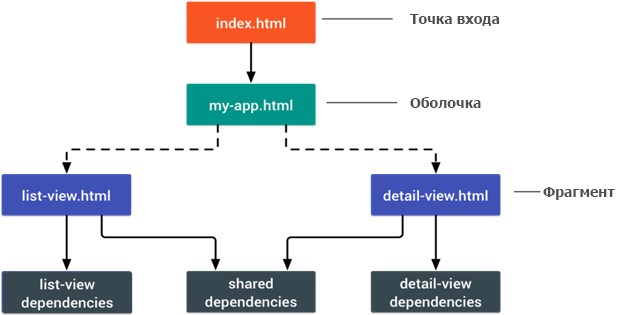
**3. Первая значимая картинка через 1.25 секунды, индекс скорости (SpeedIndex) меньше 1000.**  
Несмотря на сложность достижения, Вашей абсолютной целью должно быть начало отрисовки страницы уже через 1 секунду и значение индекса скорости ([SpeedIndex](https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index)) менее 1000 (на высокоскоростном подключении). Максимально-допустимая длительность отрисовки первого значимого изображения – 1250 миллисекунд. Для мобильных устройств отличным показателем будет начало отрисовки страницы меньше чем через 3 секунды для 3G подключения ([acceptable](https://www.soasta.com/blog/google-mobile-web-performance-study/)). Если Ваши показатели слегка перевалили за это значения – ничего страшного, но старайтесь понизить эти показатели как можно сильнее.

**ОПРЕДЕЛИМСЯ С ОКРУЖЕНИЕМ**

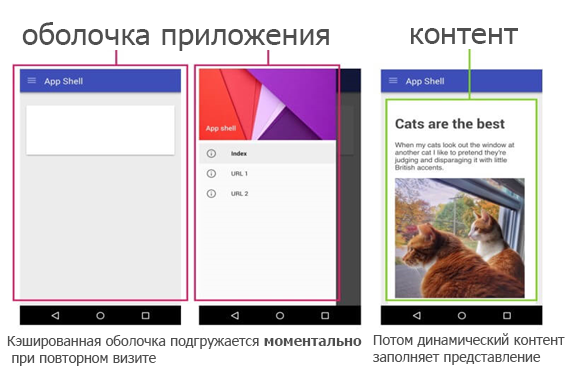
**4. Выберите для себя и сконфигурируйте инструменты сборки.**  
Не стоит слишком зацикливаться на том, что считается трендовым инструментом в данный момент. Придерживайтесь в выборе собственных интересов в разработке, не важно будет ли это Grunt, Gulp, Webpack, PostCSS или комбинация инструментов. До тех пор, пока Вам удается получать достичь результатов быстро, и Вы не ощущаете проблем в процессе использования инструментов сборки – у Вас должно быть все хорошо.

**5. Прогрессивное улучшение.**  
Смело делайте ставку на прогрессивное улучшение в качестве ведущего принцип архитектуры вашего front-end’а ([progressive enhancement](https://www.aaron-gustafson.com/notebook/insert-clickbait-headline-about-progressive-enhancement-here/)). Сначала спроектируйте и реализуйте базовые возможности, и только потом, дополняйте их сложными фичами для подходящий браузеров, создавая отказоустойчивый интерфейс ([resilient](https://resilientwebdesign.com/)). В том случае если ваш вебсайт работает быстро на медленном устройстве, со слабым браузером и посредственным интернетом, вы точно можете быть уверены, что он будет отлично работать на устройстве побыстрее, с современным браузером и хорошим интернет соединением.

**6. Angular, React, Ember и компания.**  
Присмотритесь к Фреймворку, который позволяет отрисовать картинку на стороне сервера. Не забудьте замерять время загрузки как в режиме отрисовки на сервере, так и на клиенте для мобильных устройств, прежде чем приступать к работе с Фреймворком. (так как эти изменения в будущем может быть невероятно сложно воплотить). Если Вы уже используете JavaScript фреймворк, убедитесь, что Ваш выбор является оптимальным и справедливым. ([is informed](https://www.youtube.com/watch?v=6I_GwgoGm1w)) and ([well considered](https://medium.com/@ZombieCodeKill/choosing-a-javascript-framework-535745d0ab90#.2op7rjakk)). Каждый из фреймворков по-своему будет влиять на конечную производительность и будет требовать индивидуальной стратегии оптимизации, поэтому Вам нужно отчетливо понимать всю подноготную выбранного фреймворка. В процессе построения веб-приложения обратите внимание на шаблон PRPL ([PRPL pattern](https://developers.google.com/web/fundamentals/performance/prpl-pattern/)) и архитектуру оболочки приложения. ([application shell architecture](https://developers.google.com/web/updates/2015/11/app-shell)).



*PRPL отвечает за выгрузку критично-важных ресурсов, Рендер изначальных маршрутов, Пре-кэширование остальных маршрутов и “ленивая” загрузка остальных маршрутов по запросу.*



*Оболочкой приложения называют (*[*application shell*](https://developers.google.com/web/updates/2015/11/app-shell)*) минимальный HTML, CSS, и JavaScript снабжающий интерфейс пользователя.*

**7. AMP от ребят из Google или Instant Articles от Facebook?**

В зависимости от Ваших приоритетов и стратегии организации рабочего процесса Вам предстоит выбирать между Google [AMP](https://www.ampproject.org/) и [Instant Articles](https://instantarticles.fb.com/) от Facebook. Конечно, вы можете добиться отличной производительности и без них, но AMP, к примеру, предлагает фреймворк с превосходной производительностью и бесплатным доступом к СDN, а Instant Articles взвинтят показатели производительности на Facebook. Вы, кстати, можете и сами заняться построением прогрессивных веб приложений ([progressive web AMPs](https://www.smashingmagazine.com/2016/12/progressive-web-amps/)).

**8. Подойдите мудро к вопросу о выборе вашей CDN**  
В зависимости от того насколько статична информация на вашем вебсайте, Вы можете попробовать использовать вынести часть контента во внешний ресурс ([static site generator](https://www.smashingmagazine.com/2015/11/static-website-generators-jekyll-middleman-roots-hugo-review/)), выгрузив ее на CDN и запрашивая оттуда статическую версию контента при необходимости, таким образом вы спасётесь от запросов к базе данных. Вы даже можете выбрать для себя статические хостинг-платформы ([static-hosting platform](https://www.smashingmagazine.com/2015/11/modern-static-website-generators-next-big-thing/)) базирующиеся на CDN, обогащая страницы вашего вебсайта интерактивностью, в качестве прогрессивного улучшения ([JAMStack](https://jamstack.org/)).

Обратите внимание, что вы можете использовать CDNы для разгрузки (распределения) динамического контента тоже. Так что не обязательно ограничиваться в использовании CDN только для статического контента. Обязательно перепроверьте выполняет ли выбранная CDN сжатие и преобразование контента, умную выгрузку HTTP/2, имеется ли возможность сборки статической и динамической части контента для страницы на стороне CDN’а (ближайший по расположению сервер для пользователя), и другие необходимые задачи.

**ОПТИМИЗАЦИЯ СБОРКИ**

**9. Установите для себя четкие приоритеты.**  
Было бы мудро, для начала ознакомиться с чем Вам придется работать. Проведите «инвентаризацию» всех своих ресурсов и контента (JavaScript, изображения, шрифты, внешние скрипты, и “увесистые” модули на странице, такие как карусели, сложная инфографика, и мультимедийный контент), и разделите их на группы.

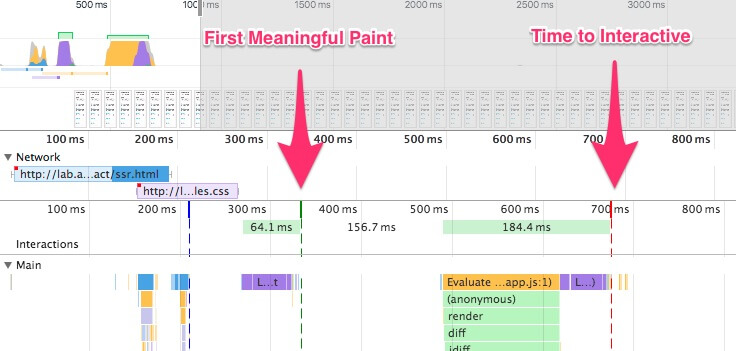
Составьте таблицу. Определите **основные моменты** для устаревших браузеров (т. е. полностью доступный основной контент), затем, **улучшенный** интерфейс для подходящих браузеров (т.е. улучшенный функционал, полный интерфейс взаимодействия) и **дополнения** (активы, который не являются абсолютно необходимыми и могут быть подгружены позже (lazy-load), такие как веб шрифты, излишние стили, скрипты каруселей, видеоплейеры, кнопки социальных сетей, крупные изображения). В качестве примера рекомендуется к рассмотрению статья (“[Improving Smashing Magazine’s Performance](https://www.smashingmagazine.com/2014/09/improving-smashing-magazine-performance-case-study/),”) которая описывает этот подход в подробностях.

**10. Используйте технику «Срезания вершков»**

Что бы развернуть главную часть интерфейса в устаревших браузерах и добавить продвинутый функционал для современных браузеров - используйте технику «Сбора вершков» ([cutting-the-mustard technique](http://responsivenews.co.uk/post/18948466399/cutting-the-mustard). Распределите процесс на стадии четко: Выгрузите главную часть моментально, улучшения – по событию DomContentLoaded и прочие дополнения по событию load.

Обратите внимание что данная техника узнает о возможностях устройства из браузера, но в последнее время, это может иметь пагубное влияние. Например, в развивающихся странах, дешевые смартфоны на Android в большинстве своем используют Chrome и смогут «собрать вершки» несмотря на их ограниченные память и возможности процессора. Знайте, что, поскольку у нас с Вами нет альтернативы данной техники использование этой техники в последнее время стало более ограниченным.

**11. Обратите внимание на микро-оптимизацию и прогрессивную загрузку.**  
В некоторых приложениях может понадобится какое-то время прежде чем вы сможет отрендерить страницу. Гораздо лучшим решением будет показывать скелет приложения ([skeleton screens](https://twitter.com/lukew/status/665288063195594752)) вместо индикаторов загрузки. Присмотритесь поближе к модулям и техникам по уменьшению скорости изначальной отрисовки страницы (Например, [tree-shaking](https://medium.com/@richavyas/aha-moments-from-ngconf-2016-part-1-angular-2-0-compile-cycle-6f462f68632e#.8b9afnsub) и  [разделение кода](https://webpack.github.io/docs/code-splitting.html)), потому что большинство времени тратится именно на этом этапе на парсинг информации для приложения. Также используйте преждевременный компилятор ([ahead-of-time compiler](https://www.lucidchart.com/techblog/2016/09/26/improving-angular-2-load-times/)) чтобы выполнить как можно более весомую часть рендера на сервере ([offload some of the client-side rendering](https://www.smashingmagazine.com/2016/03/server-side-rendering-react-node-express/)) ( [server](http://redux.js.org/docs/recipes/ServerRendering.html) ) и следовательно выведите используемые данные быстро. В конце концов возьмите на вооружение [Optimize.js](https://github.com/nolanlawson/optimize-js) для ускоренной по средствам обертывания часто-используемых функций загрузки (это может быть и не нужно [might not be necessary](https://twitter.com/tverwaes/status/809788255243739136) ).

*Прогрессивная загрузка* ([*Progressive booting*](https://aerotwist.com/blog/when-everything-is-important-nothing-is/))*подразумевает использование отрисовки страницы на сервере что бы получить первую значимую картинку максимально быстро, а также подключите минимальный JavaScript, чтобы привести время интерактивности ближе к времени отрисовки первой значимой картинки.*

Рендер на клиенте или на сервере? В обоих случая нашей целью должно быть добиться прогрессивной загрузки ([progressive booting](https://aerotwist.com/blog/when-everything-is-important-nothing-is/)): Используйте рендер на стороне сервера что бы получить первую значимую картинку как можно быстрее, а так же добавьте немного JavaScript’а что бы привести время интерактивности к времени первой значимой картинки. А догрузить (или по запросу или когда позволит время) менее значимые функции страницы мы всегда успеем. К сожалению, как заметил Пол Льюис ([Paul Lewis noticed](https://aerotwist.com/blog/when-everything-is-important-nothing-is/#which-to-use-progressive-booting)), в большинстве своем фреймворки не берут во внимание понятие приоритетности которое можно было бы использовать разработчику, и поэтому прогрессивную загрузку довольно сложно имплементировать с большинством библиотек и фреймворков. Если Вам позволяет время и ресурсы используйте эту стратегию что бы достичь идеальной загрузки.

**12. Правильно ли настроены заголовки HTTP?**

Обязательно перепроверьте что expires, cache-control, max-age и другие заголовки кэша HTTP правильно выставлены. В общем, ресурсы должны подлегать кэшированию или на очень короткий срок (если они могут изменяться) или навсегда (если они неизменны) — вы можете просто изменять их версию в URL при необходимости.

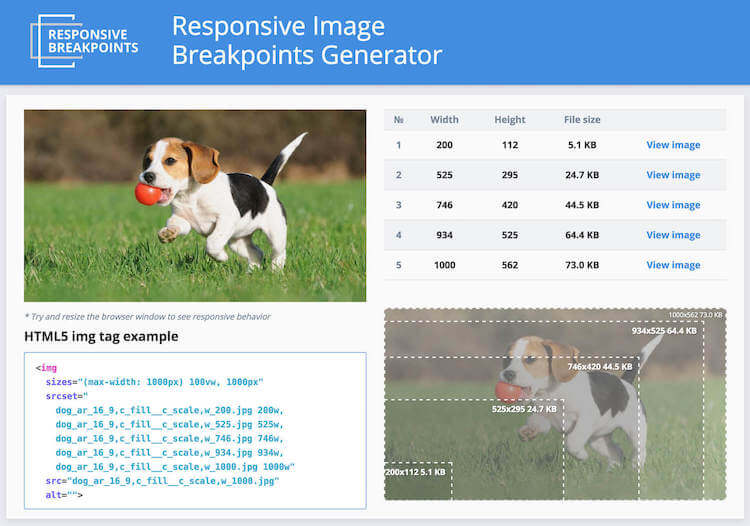
По возможности, используйте Cache-control: immutable, предназначенный для защищенных отпечатком пальца статических ресурсов, во избежание ре-валидации (на момент написания – Декабрь 2016 - поддерживается только в Firefox ([supported only in Firefox](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control)) по транзакциям https:// ). Вы можете использовать праймер заголовков HTTP кэша Хироку ([Heroku’s primer on HTTP caching headers](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers)), «Лучшие практики по кешированию» Джейка Арчибальда (“[Caching Best Practices](https://jakearchibald.com/2016/caching-best-practices/)”) и Праймер кэширования HTTP от Ильи Григорика ([HTTP caching primer](https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en)) в качестве руководства.

**13. Ограничьте сторонние библиотеки и загружайте JavaScript асинхронно.**  
Когда пользователь запрашивает страницу, браузер извлекает HTML и строит DOM, потом собирает CSS и строит CSSOM, и только потом генерирует дерево рендера, в соответствии с DOM и CSSOM. Если какой-либо скрипт должен отработать, по умолчанию, браузер не начинает рендер страницы пока скрипт не закончит работу, что вызывает задержки рендера. Нам, разработчикам, приходится настойчиво указывать браузеру что не стоит ждать окончания выполнения скриптов для начала рендера страницы. Простейшим способом добиться этого являются HTML атрибуты defer и async.

На практике, хорошо бы нам отдавать предпочтение ([prefer ) defer (to) перед  async](http://calendar.perfplanet.com/2016/prefer-defer-over-async/) (как поблажку для пользователей IE9 и старше ([cost to users of Internet Explorer](https://github.com/h5bp/lazyweb-requests/issues/42)), потому что скорее всего иначе Вы сломаете им все скрипты). Также постарайтесь ограничить влияние сторонних скриптов и библиотек, особенно кнопок социальных сетей и врезок <iframe>  (карт, например). Вместо этого вы можете использовать статические кнопки социальных сетей ([static social sharing buttons](https://www.savjee.be/2015/01/Creating-static-social-share-buttons/)) (например, [SSBG](https://simplesharingbuttons.com/)) и статические ссылки на интерактивные карты ([static links to interactive maps](https://developers.google.com/maps/documentation/static-maps/intro)).

**14. Правильно ли оптимизированы изображения?**

Как можно больше используйте адаптивные изображения ([responsive images](https://www.smashingmagazine.com/2014/05/responsive-images-done-right-guide-picture-srcset/)) с атрибутами srcset, sizes и элементом <picture>. While you’re at it, you could also make use of the [WebP format](https://www.smashingmagazine.com/2015/10/webp-images-and-performance/) by serving WebP images with the <picture> element and a JPEG fallback (see Andreas Bovens’ [code snippet](https://dev.opera.com/articles/responsive-images/#different-image-types-use-case)) or by using content negotiation (используя заголовки Accept). Sketch нативно поддерживает WebP, и изображения в формате WebP могут быть экспортированы из Photoshop используя плагин WebP для Photoshop ([WebP plugin for Photoshop](http://telegraphics.com.au/sw/product/WebPFormat#webpformat)).  Конечно, возможны и другие варианты. ([Other options are available](https://developers.google.com/speed/webp/docs/using)).



*Генератор брейкпоинтов для адаптивных изображений (* [*Responsive Image Breakpoints Generator*](http://www.responsivebreakpoints.com/)*) автоматизирует создание изображений и разметки.*

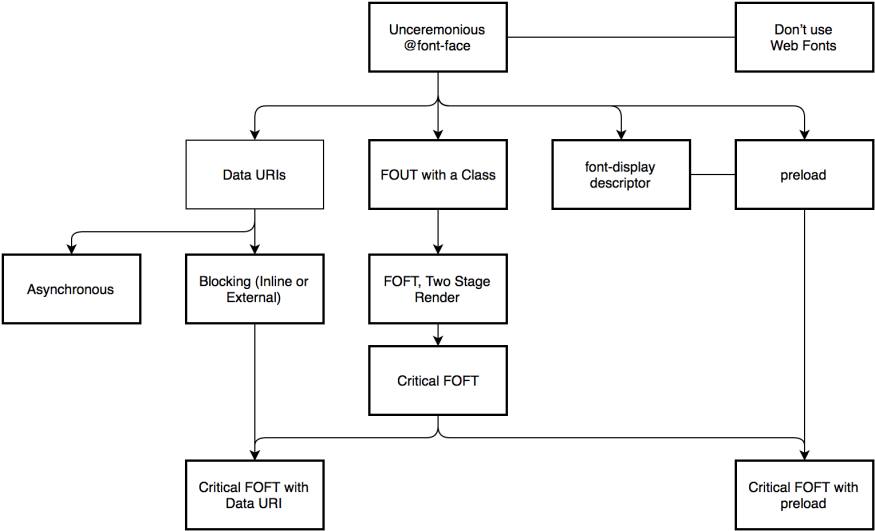
Вы также могли бы использовать клиентские подсказки, ([client hints](https://www.smashingmagazine.com/2016/01/leaner-responsive-images-client-hints/)), они как раз сейчас заручаются поддержкой браузеров ([gaining browser support](http://caniuse.com/#search=client-hints)). Не успеваете заняться сложной разметкой для адаптивности Ваших изображений? Вам в помощь Генератор *брейкпоинтов для адаптивных изображений (* [*Responsive Image Breakpoints Generator*](http://www.responsivebreakpoints.com/)*)* а также другие сервисы автоматизированной оптимизации изображений, например [Cloudinary](http://cloudinary.com/documentation/api_and_access_identifiers). Также, в большинстве случаев, использование одних только srcset и sizes принесет значительную выгоду. Например, ребята из Smashing Magazine, используют -opt для имён изображений — например, brotli-compression-opt.png; Как только изображение содержит в себе постфикс opt, каждый член команды знает, что его уже не нужно оптимизировать.

**15. Выведите оптимизацию изображений на новый уровень.**  
Когда Вы работаете над лэндингом, для которого критичным является молниеносная загрузка некоторых изображений убедитесь что Ваши JPEGи прогрессивны и сжаты с помощью  [mozJPEG](https://github.com/mozilla/mozjpeg) (который улучшает скорость начала рендера изображения с помощью манипуляций с уровнями сканирования), [Pingo](http://css-ig.net/pingo) для PNG, [Lossy GIF](https://kornel.ski/lossygif) для GIF и [SVGOMG](https://jakearchibald.github.io/svgomg/) для SVG. Размойте ненужные части изображения (применяя к ним размытие по Гауссу) что бы уменьшить размер изображения, если даже это не помогает, в конце концов можете попробовать лишать изображение цвета, черно-белое изображение весит гораздо меньше. Для фоновых изображений вполне приемлемым будет экспорт изображений с качеством от 0 до 10%.

Все еще не довольны результатом? Ну тогда, Вам придется обратиться к технике множественных фоновых изображений ([multiple](http://csswizardry.com/2016/10/improving-perceived-performance-with-multiple-background-images/) [background](https://jmperezperez.com/medium-image-progressive-loading-placeholder/) [images](https://manu.ninja/dominant-colors-for-lazy-loading-images#tiny-thumbnails) [technique](https://css-tricks.com/the-blur-up-technique-for-loading-background-images/)).

**16. Как обстоят дела с оптимизацией веб шрифтов?**

Велик шанс, что в веб-шрифтах которые вы используете в своем проекте присутствуют символы, которые вы не используете. Вы могли бы попросить поставщика шрифтов сузить набор символов для ваших нужд или заняться этим самостоятельно, ( [subset them yourself](https://www.fontsquirrel.com/tools/webfont-generator) ) Если Вы используете открытые шрифты (например, включая в алфавит только некоторые символы из латиницы) чтобы уменьшить размеры файла. Поддержка [WOFF2 support](http://caniuse.com/#search=woff2) великолепна, а для подстраховки Вы можете использоваться WOFF и OTF специально для браузеров с отстающей поддержкой. Также выберите для себя одну из стратегий Зака Лэзермана – “[Полное руководство по стратегиям загрузки шрифтов](https://www.zachleat.com/web/comprehensive-webfonts/),” а также используйте кэш сервис воркеров для постоянного кэширования шрифтов. Хотите быстрых результатов? У Pixel Ambacht есть краткое исследование вопроса ([quick tutorial and case study](https://pixelambacht.nl/2016/font-awesome-fixed/)), как привести Ваши шрифты в порядок.



*“*[*Полное руководство по стратегиям загрузки шрифтов*](https://www.zachleat.com/web/comprehensive-webfonts/)*,” Зака Лэзермана предоставляет дюжену опций для улучшения выгрузки шрифтовprovides a dozen of options for better web font delivery.*

Обязательно используйте [Web Font Loader](https://github.com/typekit/webfontloader) если Вам приходится использовать внешние сервера для загрузки шрифтов. [FOUT лучше чем FOIT](https://www.filamentgroup.com/lab/font-events.html); сразу же начните отрисовку текста, и загрузите шрифты асинхронно— можете использовать [loadCSS](https://github.com/filamentgroup/loadCSS) для этих целей. Вы так же обойтись и шрифтами установленными локально в ОС ([get away with locally installed OS fonts](https://www.smashingmagazine.com/2015/11/using-system-ui-fonts-practical-guide/)).

**17. Молниеносно выгрузите критичные CSS стили.**  
Что бы быть уверенным на все 100% что Ваш браузер начнет рендер страницы настолько быстро, на сколько это возможно, общепринятой практикой ([common practice](https://www.smashingmagazine.com/2015/08/understanding-critical-css/) ) становится собирать весь CSS необходимый для начала рендера первой видимой части страницы (известный как «критично-важный» CSS) и добавлять его инлайново в &lthead&gt  вашего документа таким образом уменьшая запросы зависимости. Из-за ограничения на передачу пакетов во время медленной фазы ~ 14 KB это принято считать Вашим бюджетом критично важных CSS. Если Вы выйдете за рамки бюджета, браузер совершать дополнительные обращения чтобы выгрузить больше стилей. [CriticalCSS](https://github.com/filamentgroup/criticalCSS) и [Critical](https://github.com/addyosmani/critical) позволяют Вам управится с задачей. Вам скорее всего понадобится проделать это с каждым шаблоном над которым Вы работаете. Если возможно, постарайтесь использовать условно-инлайновый подход ([conditional inlining approach](https://www.filamentgroup.com/lab/performance-rwd.html))  представленный Filament Group.

С появлением HTTP/2, критично-важный CSS можно хранить в отдельном файле CSS и выгружать по запросу с сервера с помощью сервер пуша не загрязняя HTML. Загвоздка в том, что сервер пуш на данный момент не поддерживается и вызывает некоторые проблемы с кэшированием (слайд 114 презентации Хумана Бэхешти [Hooman Beheshti’s presentation](http://www.slideshare.net/Fastly/http2-what-no-one-is-telling-you)). Результат может быть негативным, и привести к загрязнению сетевых буферов ([be negative](http://calendar.perfplanet.com/2016/http2-push-the-details/)), и предотвращению доставки истинных пакетов. Сервер пуш гораздо более эффективен на «теплом» соединении ([more effective on warm connections](https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/edit))  из-за медленного старта TCP. Так что Вам может понадобиться создать механизм HTTP/s сервер пуша осведомленный о кэше( [cache-aware HTTP/2 server push mechanism](https://css-tricks.com/cache-aware-server-push/)). Помните, что новая спецификация [cache-digest specification](http://calendar.perfplanet.com/2016/cache-digests-http2-server-push/) снизит потребность в создании таких механизмов вручную.

**18. Используйте tree-shaking и code-splitting для уменьшения нагрузки.**

[Tree-shaking](https://medium.com/@roman01la/dead-code-elimination-and-tree-shaking-in-javascript-build-systems-fb8512c86edf) это способ расчистить ваш процесс сборки вовлекая только ту часть кода, которая действительно используется на продакшене. Вы можете пользоваться Webpack 2 что бы свести к минимум ненужные экспорты кода, ([use Webpack 2 to eliminate unused exports](http://www.2ality.com/2015/12/webpack-tree-shaking.html)), и [UnCSS](https://github.com/giakki/uncss) или [Helium](https://github.com/geuis/helium-css) что бы предотвратить добавление в билд неиспользуемых стилей. Также Вы могли бы научиться писать эффективные CSS селекторы и избегать нагромождения весомых стилей.  ([write efficient CSS selectors](http://csswizardry.com/2011/09/writing-efficient-css-selectors/)) ([avoid bloat and expensive styles](https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/)).

[Code-splitting](https://webpack.github.io/docs/code-splitting.html) это еще одна фича Webpack’а, которая разбивает основание кода на куски и подгружает по запросу. Как только Вы определите точки разделения вашего кода, Webpack сможет позаботиться о зависимостях, и выходных файлах. Он, по сути, позволяет тебе держать изначальный трафик минимальным и запрашивать код при необходимости самим приложением.

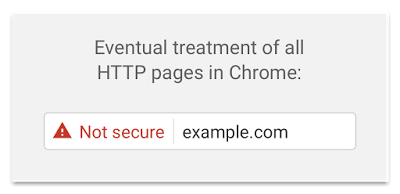
Примите к сведению что [Rollup](http://rollupjs.org/) показывает гораздо лучшие результаты чем выдает Browserify. Раз уж мы заговорили от этом, вам может быть также интересен [Rollupify](https://github.com/nolanlawson/rollupify), который превращает модули ECMAScript 2015 в один большой CommonJS модуль — потому что маленькие модули могут требовать, на удивление, больших затрат производительности, ([surprisingly high performance cost](https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/)) в зависимости от сборщика и модульной системы, которой вы пользуетесь.

**19. Улучшайте производительность рендера.**  
Изолируйте увесистые элементы с помощью политики сдерживания [CSS](http://caniuse.com/#search=contain)  — например, что бы ограничить стили браузера для компоновки и окраса элементов, для навигации вне канваса, или для виджетов третьей стороны. Удостоверьтесь что нет промедлений при прокрутке страницы и при анимации элементов, и что пользователь сможет получать стабильные 60 кадров в секунду. Если это не является возможным – как минимум предоставьте диапазон значений FPS в рамках 15-60. Используйте свойство CSS [will-change](http://caniuse.com/#feat=will-change) что бы сообщить браузеру, какие элементы и свойства в будущем изменятся.

Также, замерьте время производительности рендера ([runtime rendering performance](https://aerotwist.com/blog/my-performance-audit-workflow/#runtime-performance)) (например, [в инструментах разработчика](https://developers.google.com/web/tools/chrome-devtools/rendering-tools/)). Перед тем как приступить, пройдите бесплатный курс на [Udacity – отпимизация рендера браузером](https://www.udacity.com/course/browser-rendering-optimization--ud860).

**20. Warm up the connection to speed up delivery.**  
Используйте скелетный экраны в интерфейсе и ленивую загрузку всех увесистых компонентов, как шрифтов fonts, JavaScript, карусели, видео и ifram’ы. Используйте ресурсные подсказки ([resource hints](https://w3c.github.io/resource-hints)) в целях сохранения времени на [dns-prefetch](http://caniuse.com/#search=dns-prefetch) (который выполняет DNS lookup в фоновом режиме), [preconnect](http://www.caniuse.com/#search=preconnect) (который просит у браузера разрешение на начало рукопожатий (DNS, TCP, TLS – так же в фоне), [prefetch](http://caniuse.com/#search=prefetch)(который просит браузер запрашивать ресурс), [prerender](http://caniuse.com/#search=prerender) (который указывает браузера как отрендерить страницу в фоновом режиме) и [preload](https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for/) (который, в том числе подготавливает ресурсы, до взаимодействия с ними). Обратите внимание, что на практике, в зависимости от поддержки браузера, вы скорее посмотрите в сторону preconnect чем dns-prefetch, и вы будете очень осторожны с prefetch и prerender — последний следует использовать только если вы крайне уверены как пользователь будет себя вести и каким будет его следующий шаг (хороший пример – воронка продаж).

#### HTTP/2

**21. Будьте готовы к HTTP/2.**  
Вместе с Google, двигающимся в сторону \*[moving towards a more secure web](https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html)\*  более безопасного веба и окончательного восприятия HTTP страниц Chrome’ом как «небезопасные» вам придется решиться сделать ставку, или на конфигурацию окружения [HTTP/2](https://http2.github.io/faq/)  или остаться на HTTP/1.1. HTTP/2 отлично поддерживается ([supported very well](http://caniuse.com/#search=http2)); Он никуда не денется и в большинстве случаев лучше бы Вам принять его сторону. Безусловно, это достаточно трудоемкий процесс, но рано или поздно, Вам придется перейти на HTTP/2. Когда Вы с этим покончите, вы получите отличный прирос в производительности с сервис воркерами и сервер пушами ([major performance boost](https://www.youtube.com/watch?v=RWLzUnESylc&t=1s&list=PLNYkxOF6rcIBTs2KPy1E6tIYaWoFcG3uj&index=25) ). [](https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html)

В итоге, Google собирается помечать все HTTP страницы как небезопасные и поменять индикатор безопасности на этих страницах на красный треугольник, такой же как сейчас у сломанных HTTPS.

Отрицательный эффект такая миграция на HTTPS, может возыметь в случае если Ваша база пользователей HTTP/1.1 очень велика, и большая часть из них использует устаревшие браузеры и операционные системы. В этом случае Вам придется отправлять разные сборки приложения, который должны будут адаптироваться под разные процессы взаимодействия( [different build process](https://rmurphey.com/blog/2015/11/25/building-for-http2). Помните: Установка как миграции так и новой сборки могут быть достаточно сложными и время затратными. Для последующих частей статьи, предположим, что Вы ли уже переключились на HTTP/2 или уже занимаетесь этим вопросом.

**22. Правильно внедряйте HTTP/2.**  
и еще раз, обработка контента через HTTP/2 [serving assets over HTTP/2](https://www.youtube.com/watch?v=yURLTwZ3ehk) требует больших доработок и сильно отличается от того, как вы обрабатывали ваши активы раньше. Вам придется найти наилучший баланс между упаковки всех модулей в один и загрузки множества мелких модулей параллельно.

С одной стороны, вам может показаться хорошей идеей избежать сливания всех активов вместе, и разделись весь Ваш интерфейс на множество маленьких модулей вместо этого, сжимая и в процессе сборки, ссылаясь на них с подходом «скаута» ( [“scout” approach](https://rmurphey.com/blog/2015/11/25/building-for-http2)) и загружая их в параллель. В отличии от одного файла в случае неудачной загрузки не придется выгружать полностью все стили и скрипты заново.

С другой стороны, [упаковка всех файлов в один](http://engineering.khanacademy.org/posts/js-packaging-http2.htm) все еще имеет место быть, потому что существуют проблемы с отправкой нескольких многих маленьких JavaScript файлов в браузерах.   
Во-первых**, пострадает сжатие**. Компрессия больших файлов изменится в лучшую сторону благодаря повторному использованию словаря, что не касается отдельных небольших файлов. Ведется работа над улучшением этого момента, но это еще далеко от реальности. Во-вторых, браузеры еще не оптимизированы **под такой ход работы с** файлами. Например, Chrome сочтет подобные изменения за меж-процессные коммуникации[inter-process communications](https://www.chromium.org/developers/design-documents/inter-process-communication) (IPCs) между каждым из кластеров, поэтому включая сотни файлов вы рискуете сильно потерять в производительности.



*Что бы достичь лучших результатов используя HTTP/2, попробуйте подгружать CSS прогрессивно (*[*load CSS progressively*](https://jakearchibald.com/2016/link-in-body/)*), как было предложено Джеком Арчибальдом из Chrome.*

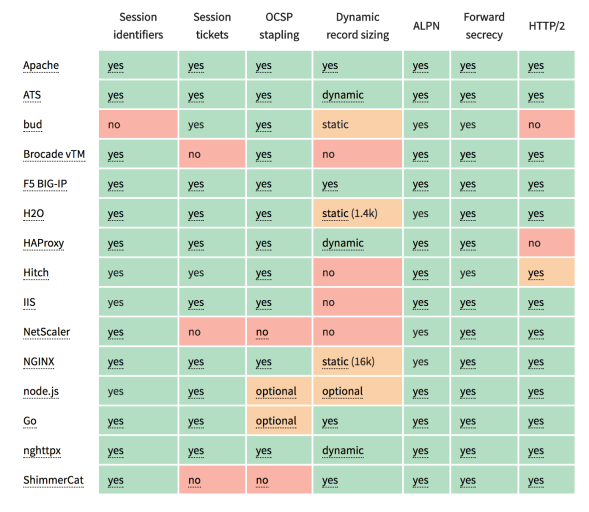
Но вы все еще можете попробовать загружать CSS прогрессивно ([load CSS progressively](https://jakearchibald.com/2016/link-in-body/)). Очевидно, что поступая таким образом вы будете активно ущемлять пользователей HTTP/1.1, так что Вам может понадобится генерация и обработка разных билдов для разных браузеров в процессе вашего деплоймента, и тут все становится немного сложнее. You could get away with [HTTP/2 connection coalescing](https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/), which allows you to use domain sharding while benefiting from HTTP/2, но на практике достичь этого сложно.

Так что же делать? Если Ваш вебсайт работает через HTTP/2, отправка около **10 сегментов** является неплохим компромиссом (и это совсем не плохо даже для устаревших браузеров). Просто поэкспериментируйте и вычислите нужный баланс для вашего вебсайта.

**23. Удостоверьтесь в пуленепробиваемой защите вашего сервера.**  
Все браузерные реализации HTTP/2 работаю через TLS, так что скорее всего Вам захочется избежать оповещений безопасности или неработоспособности каких-то элементов на вашей странице. Перепроверьте что заголовки безопасности правильно настроены, ([security headers are set properly](https://securityheaders.io/)), избавьтесь от заранее известных уязвимостей([eliminate known vulnerabilities](https://www.smashingmagazine.com/2016/01/eliminating-known-security-vulnerabilities-with-snyk/)), и проверьте Ваш сертификат ([check your certificate](https://www.ssllabs.com/ssltest/)).

Все еще не перешли на HTTPS? Присмотритесь к HTTPS-Only Стандарту в качестве инструкции. ([The HTTPS-Only Standard](https://https.cio.gov/faq/)). Также, удостоверьтесь что внешние плагины и отслеживающие скрипты загружены через HTTPS, что межсайтовый скриптинг невозможен и что как строгие заголовки транспортной безопасности HTTP, так и заголовки политики безопасности подключений заданы верно. ([HTTP Strict Transport Security headers](https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet)) ([Content Security Policy headers](https://content-security-policy.com/)).

**24. Поддерживают ли Ваш сервер и CDNы HTTP/2?**  
Разные сервера и CDNы скорее всего будет поддерживать HTTP/2 по-разному. Попробуйте [Is TLS Fast Yet?](https://istlsfastyet.com/) Что-бы уточнить информацию касательно вашего варианта или в сравнении с другими узнать насколько производителен Ваш сервер и поддержку каких фич стоит ожидать в скором времени.

[](https://istlsfastyet.com/)

Ресурс [Is TLS Fast Yet?](https://istlsfastyet.com/) Позволяет Вам проанализировать конфигурации серверов и CDN при переходе на HTTP/2.

**25. Используется ли сжатие Brotli или Zopfli?**  
В прошлом году Google представили миру Brotli ([introduced](https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html) [Brotli](https://github.com/google/brotli)), новый формат данных без потерь, который уже широко поддерживается ([widely supported](http://caniuse.com/#search=brotli)) в Chrome, Firefox и Opera. На практике, Brotli представляется более эффективным ([more effective](https://samsaffron.com/archive/2016/06/15/the-current-state-of-brotli-compression)) чем Gzip и Deflate. Сам процесс сжатия может быть затяжным, в зависимости от настроек и чем дольше длится сжатие, тем выше будет уровень самого сжатия. Кстати говоря, декомпрессия происходит быстро. Не будет сюрпризом что браузеры будут поддерживать этот алгоритм только если пользователь посещает вебсайт через HTTPS, ведь алгоритм разработан в Google, но на самом деле, на то есть еще и технические причины. Загвоздка в том, что Brotli не предустановлен на большинстве серверов сегодня, и не так уж просто его настроить без самостоятельной компиляции NGINX или Ubuntu. Однако, вы можете активировать Brotli даже на тех CDN, которые пока что его не поддерживают ([enable Brotli even on CDNs that don’t support it](http://calendar.perfplanet.com/2016/enabling-brotli-even-on-cdns-that-dont-support-it-yet/)) yet (с помощью сервис воркеров).

В качестве альтернативы, можете рассмотреть алгоритм сжатия Zopfli ([Zopfli’s compression algorithm](https://blog.codinghorror.com/zopfli-optimization-literally-free-bandwidth/)), который кодирует информацию в форматы Deflate, Gzip и Zlib. Любой ресурс сжатый простым Gzip-ом можно выгодно сжимать с Zopfli’s с улучшеным кодированием Deflate, потому что файлы будут на 3-8% меньше чем может предоставить лушчее сжатие от Zlib’s. Загвоздка в тому, что это займет гораздо больше времени (~в 80 раз дольше). Поэтому использование Zopfli будет отличным решением для ресурсов, которые не очень-то меняются, файлы, которые, были созданы, чтобы однажды быть сжатыми и множество раз скачанными.

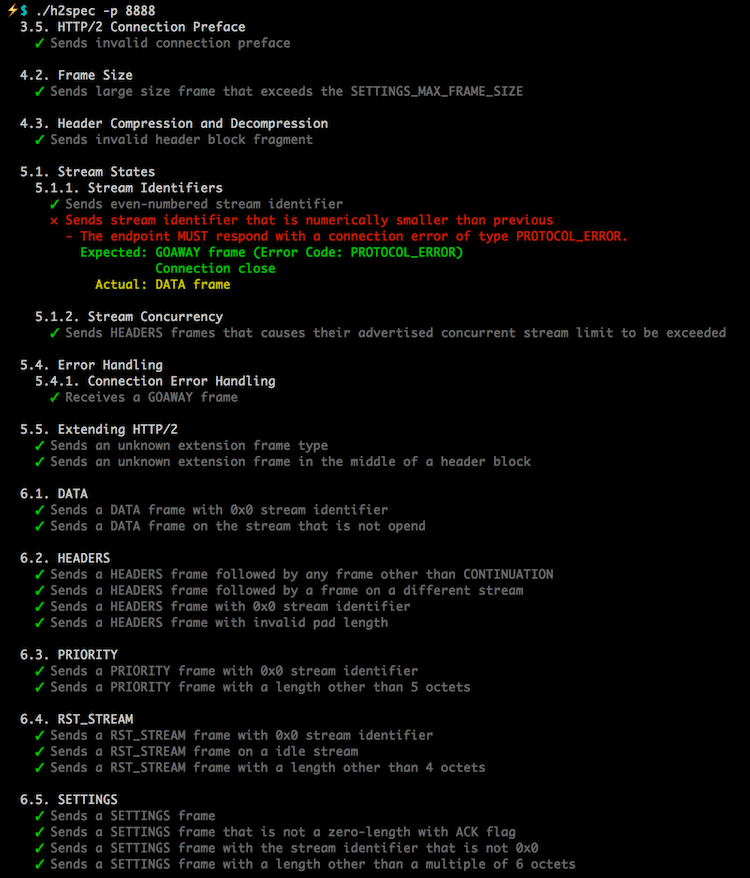
**26. Активировано ли OCSP сшивание?**

Включая сшивание OCSP на своем сервере ([enabling OCSP stapling on your server](https://www.digicert.com/enabling-ocsp-stapling.htm)), Вы сможете значительно ускорить «рукопожатия» протокола TLS. Протокол OCSP (Online Certificate Status Protocol) был создан в качестве альтернативы для протокола CRL (Certificate Revocation List). Оба протокола используются для проверки валидности SSL сертификата. Однако, протокол OCSP не требует чтобы браузер тратил время, загружая список сертификатов и детальную информацию про них для поиска, таким образом уменьшая время затраченное на «рукопожатие».

**27. Успели ли Вы взять IPv6 на вооружение?**  
Поскольку запасы адресов IPv4 подходят к концу ([running out of space with IPv4](https://en.wikipedia.org/wiki/IPv4_address_exhaustion))  , большинство мобильных сетей получают поддержку IPv6 просто молниеносно (США достигли порога в 50% поддержки IPv6 [reached](https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption)), будет замечательной идеей обновить поддержку Вашего DNS на IPv6 ([update your DNS to IPv6](https://www.paessler.com/blog/2016/04/08/monitoring-news/ask-the-expert-current-status-on-ipv6) ) что бы в будущем обеспечить себе спокойный сон. Просто убедитесь что во всей сети предоставляется двойной стэк – IPv4 и IPv6 — это позволит IPv6 и IPv4 работать одновременно бок о бок. В конце концов, IPv6 не обратно-совместимый. Также, исследования показали ( [studies show](https://www.cloudflare.com/ipv6/) ) что IPv6 сделала эти сайты на 10-15% быстрее благодаря техника обнаружения соседа (NDP) и оптимизации маршрута.

**28. Пользуетесь ли Вы HPACK сжатием?**

В случае если вы используете HTTP/2, стоит убедиться, что Ваши сервера используют сжатие HPACK ([implement HPACK compression](https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/)) в заголовках HTTP ответов, чтобы уменьшить использование ресурсов. Из-за того, что сервера HTTP/2 относительно молоды, может иметь место неполноценная поддержка спецификаций, с HPACK, например. [H2spec](https://github.com/summerwind/h2spec) это отличный (и очень детализированный) инструмент что бы проверить работоспособность HPACK ([HPACK works](https://www.keycdn.com/blog/http2-hpack-compression/)).

[](https://www.smashingmagazine.com/wp-content/uploads/2016/12/h2spec-example-large-opt.png)  
H2spec

**29. Работают ли сервис воркеры для кэширования и сетевых фолбэков?**

Как ни оптимизируй сетевую передачу данных, она не будет быстрее чем получение данных из локального кэша, хранящегося на устройстве пользователя. Если Ваш вебсайт использует HTTPS, стоит присмотреться к «Прагматичному гиду по сервис воркерам» (“[Pragmatist’s Guide to Service Workers](https://github.com/lyzadanger/pragmatist-service-worker)”) что бы добавлять в кэш сервис воркеров статические материалы и локально хранить офлайн фолбэки (или целые офлайн страницы) и открывать их из памяти устройства пользователя а не отправляться за ними в. Будет полезным также посмотреть «Книгу рецептов для Офлайна» от Джейка ([Offline Cookbook](https://jakearchibald.com/2014/offline-cookbook/)) и бесплатный Udacity курс “[Offline Web Applications](https://www.udacity.com/course/offline-web-applications--ud899).” Волнуетесь о поддержке браузерами? Не стоит, она на подходе ([getting there](http://caniuse.com/#search=serviceworker)), да и сеть в любом случае останется запасным вариантом.

#### Тестирование и мониторинг

**30. Отслеживайте предупреждения *смеси-контента* (mixed-content).**

Если Вы не так давно перешли с HTTP на HTTPS, не забудьте промониторить предупреждения микс-контента, как активного так и пассивного, для этого подойдёт инструмент [Report-URI.io](https://report-uri.io/). Вы также можете использовать [Mixed Content Scan](https://github.com/bramus/mixed-content-scan) что бы просканировать Ваш вебсайт с активированным HTTPS-на содержание микс-контента.

**31. Оптимизирован ли Ваш рабочий процесс с Инструментами разработчика (DevTools)?**

Выберите для себя инструменты отладки и прокликайте каждую кнопку, ссылку. Убедитесь, что Вы знаете, как проанализировать и улучшать производительность отрисовки страницы, консольный вывод, и, как отладить JavaScript или корректировать CSS стили. Не так давно Умар Ганза (Umar Hansa) подготовил огромную презентацию ([slidedeck](https://umaar.github.io/devtools-optimise-your-web-development-workflow-2016/#/)) и речь ([talk](https://www.youtube.com/watch?v=N33lYfsAsoU)), в которых упоминаются дюжины неизведанных советов и техник о которых не стоит забывать, когда дело доходит до отладки и тестирования в Инструментах разработчика (DevTools).

**32. Какие результаты тестирования в устаревших браузерах? В прокси браузерах?**

Тестирования в Chrome и Firefox, однозначно недостаточно. Оцените, как Ваш вебсайт выглядит в прокси-браузере и в устаревших браузерах. К примеру, UC Browser и Opera Mini, занимают огромный сегмент рынка в Азии ([significant market share in Asia](http://gs.statcounter.com/#mobile_browser-as-monthly-201511-201611)) (до 35% в Азии). ([Measure average Internet speed](https://www.webworldwide.io/)) Исследуйте и замерьте среднюю скорость подключения к сети в странах, на которых ориентирован Ваш продукты, чтобы в будущем не столкнуться с неприятным сюрпризом. Не забудьте провести тестирование с троттлинг и симулируйте дисплей повышенной плотности. [BrowserStack](https://www.browserstack.com/) - это отличный инструмент, но тестирование на физических девайсах не менее.

**33. Вы установили непрерывный мониторинг?**

Всегда гораздо выгоднее иметь собственный экземпляр [WebPagetest](http://www.webpagetest.org/)'а для быстрых и неограниченных тестирований. Установите непрерывный мониторинг за бюджетом производительности с автоматическими извещениями. Установите собственные временные оценки что бы измерять, сравнивать и мониторить метрики в зависимости от сферы деятельности. Присмотритесь к использованию [SpeedCurve](https://speedcurve.com/) чтобы отслеживать изменения в производительности со временем, и/или [New Relic](https://newrelic.com/browser-monitoring) что бы получить выводы, которые *WebPagetest* не предоставляет. Стоит так же обратить внимание на [SpeedTracker](https://speedtracker.org/), [Lighthouse](https://github.com/GoogleChrome/lighthouse) и [Calibre](https://calibreapp.com/).

### Моментальный результат!

Список достаточно обширный, и выполнение оптимизация может отнять у Вас порядочно времени. А что если бы у Вас был лишь 1 час что бы добиться значительных улучшений? Какое бы решение приняли Вы? Давайте сузим список до десятки самых просто-достижимых целей. Очевидно, Вам придется замерять результаты до начала и после окончания оптимизации, включая время начала рендера и Индекс скорости на 3G и проводном подключении.

1. Вашей целью является начало отрисовки страницы не позже чем через 1 секунду на проводном подключении и 3 секунды используя 3G, и индекс скорости величиной менее 1000. Оптимизируйте время начала отрисовки и время интерактивности.
2. Подготовьте критичный CSS для основных шаблонов, и включите их в <head> вашей страницы. *(Ваш бюджет - 14 KB).*
3. Используйте *Defer* и *lazy-load* во всех скриптах, где только возможно, как в своих собственных, так и в сторонних — особенно это касается социальных медиа ссылок, видеоплейеров, и увесистого JavaScript.
4. Добавьте подсказки ресурсов, чтобы ускорить загрузки с быстрым dns-lookup, preconnect, prefetch, preload и prerender.
5. Разделите веб шрифты и загружайте их асинхронно (или просто переключайтесь на системные шрифты, как альтернатива).
6. Оптимизируйте изображения, рассмотрите возможность использования WebP для основных страниц (например, лэндингов).
7. Удостоверьтесь что сами заголовки кэша HTTP и их безопасность установлены правильно.
8. Активируйте сжатие Brotli или Zopfli на сервере. (В случае если это невозможно, не забывайте про сжатие Gzip.)
9. Если HTTP/2 доступен, активируйте сжатие HPACK и начните отслеживать предупреждения смешанного контента. Если вы работаете над LTS, также активируйте сшивание OCSP.
10. Если представляется возможным, кэшируйте содержимое, например, шрифты, стили, скрипты и изображения, а вообще, чем больше – тем лучше! — в кэши сервис воркеров.

### Скачайте чеклист для себя(PDF)

Держа в голове этот чеклист, вы гарантированно будете подготовлены к разработке любого front-end проекта в 2017 году. Не забывайте скачать PDF файл с чеклистом, подготовленным для печати для Ваших нужд:

* [Download the checklist PDF](http://provide.smashingmagazine.com/performance-checklist/performance-checklist-1.0.pdf) (PDF, 0.129 MB ENG)
* [Скачать](http://provide.smashingmagazine.com/performance-checklist/performance-checklist-1.0.pdf) чеклист в PDF на русском (PDF, 0.129 MB RUS)

Если Вас интересуют альтернативные варианты чеклиста, Вы так же можете обратить внимание на работы Дэна Рублика ([front-end checklist by Dan Rublic](https://github.com/drublic/checklist)) и Джона Яблонски (“[Designer’s Web Performance Checklist](http://jonyablonski.com/designers-wpo-checklist/)”).

### Ну что ж, поехали!

Некоторая часть процесса оптимизации может выходить за рамки вашего проекта и/или бюджета, а также быть попросту невыполнимым заданием, работая, скажем, с доисторическим кодом, который Вам достался. Это вполне нормально! Рекомендуется использовать данный чеклист как общие (и, надеюсь, фундаментальные) инструкции, или создать на его основе свой собственный список тем, которые будут применимы для Вашего проекта. Кстати, не забывайте протестировать и замерить результаты Ваших проектов до начала оптимизации, чтобы определить наиболее важные моменты. Удачной всем оптимизации в новом, 2017 году!