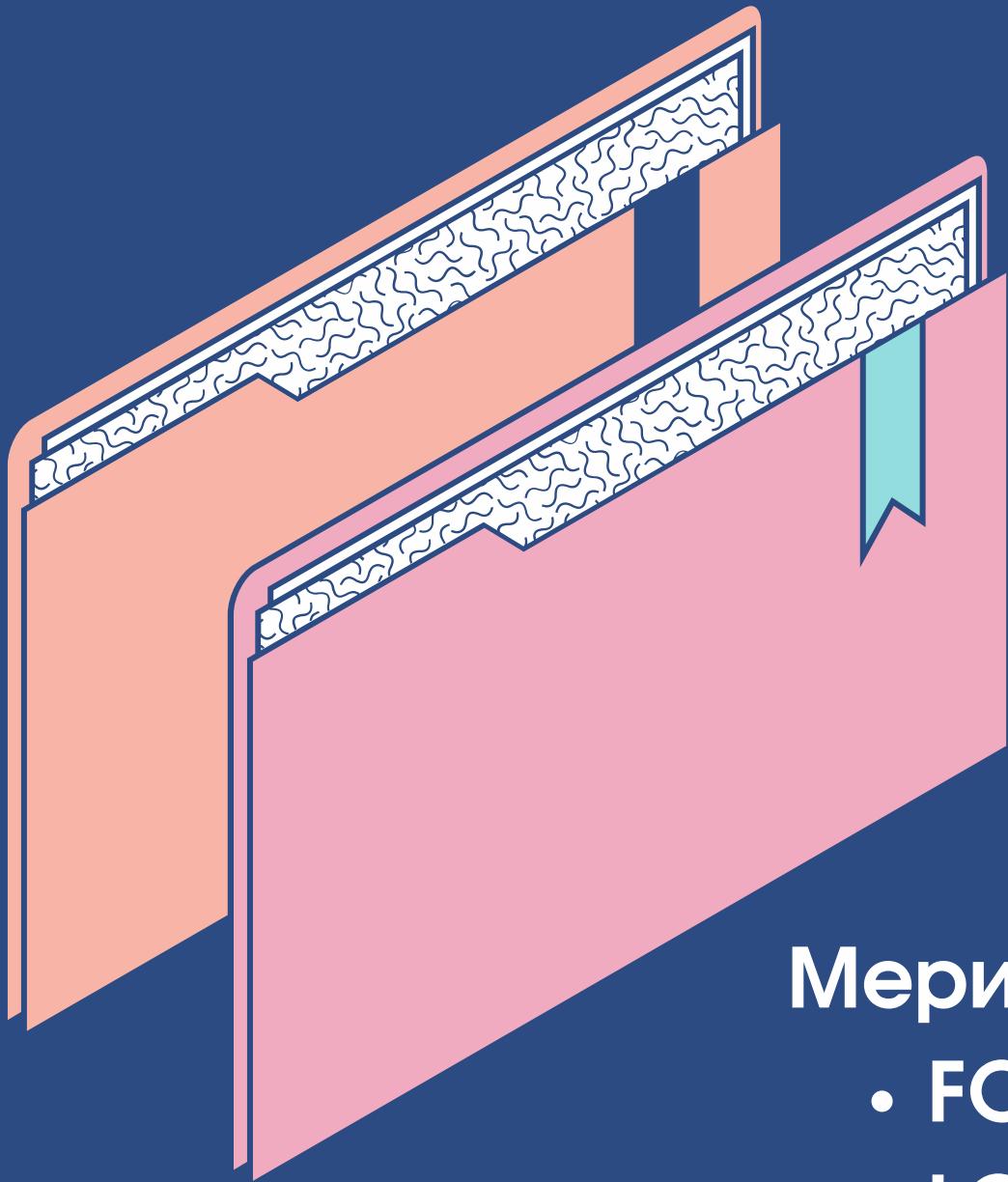




НИКОЛАЙ КОШКАРЁВ

# Оптимизация производительности ssr-приложений

Скажу коротко: метрики производительности важны.



# Какие метрики?

**Мерилом выберем PageSpeed.** Расшифровка аббревиатур:

- **FCP** - через сколько пользователь увидит хоть что-нибудь.
- **LCP** - когда загрузился самый большой видимый контент
- **TTI** - time to interactive (не знаю что еще добавить)
- **TBT** - total blocking time
- **CLS** - сдвиг контента после полной загрузки

# Какие метрики?

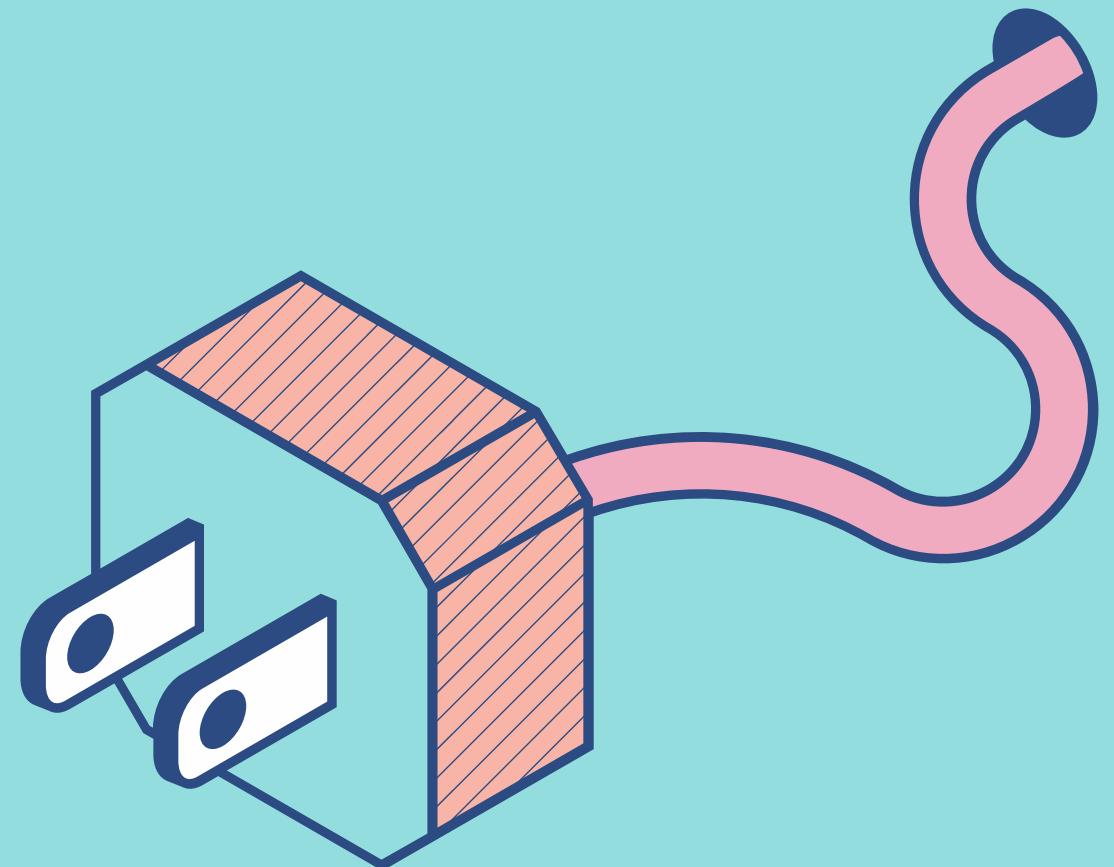
Современный фронт не мыслит себя без **spa**.

Несколько но:

- SEO
- LCP, FCP, TBT, TTI - JS...

Тут нам на помощь спешит еще одна новомодная технология (уже нет): server side rendering.

Покажу несколько техник,  
которые позволяют хоть как то  
решить эти проблемы.





# Что наше супер приложение делает

- Измерять будем для мобилок
- Для тестов построим приложение на базе фреймворка пихт.js (работает и на react.js)
- Приложение - 1 страница из шапки и 10 секций. Каждая показывает 10 карточек с товарами (сначала были котики).
- Данные каждая секция запрашивает самостоятельно на бекенде в методе fetch.
- Так же на сервере есть запрос за категориями, из которых будем строить меню.



## *Для тех кто не пользовался nuxt:*

”

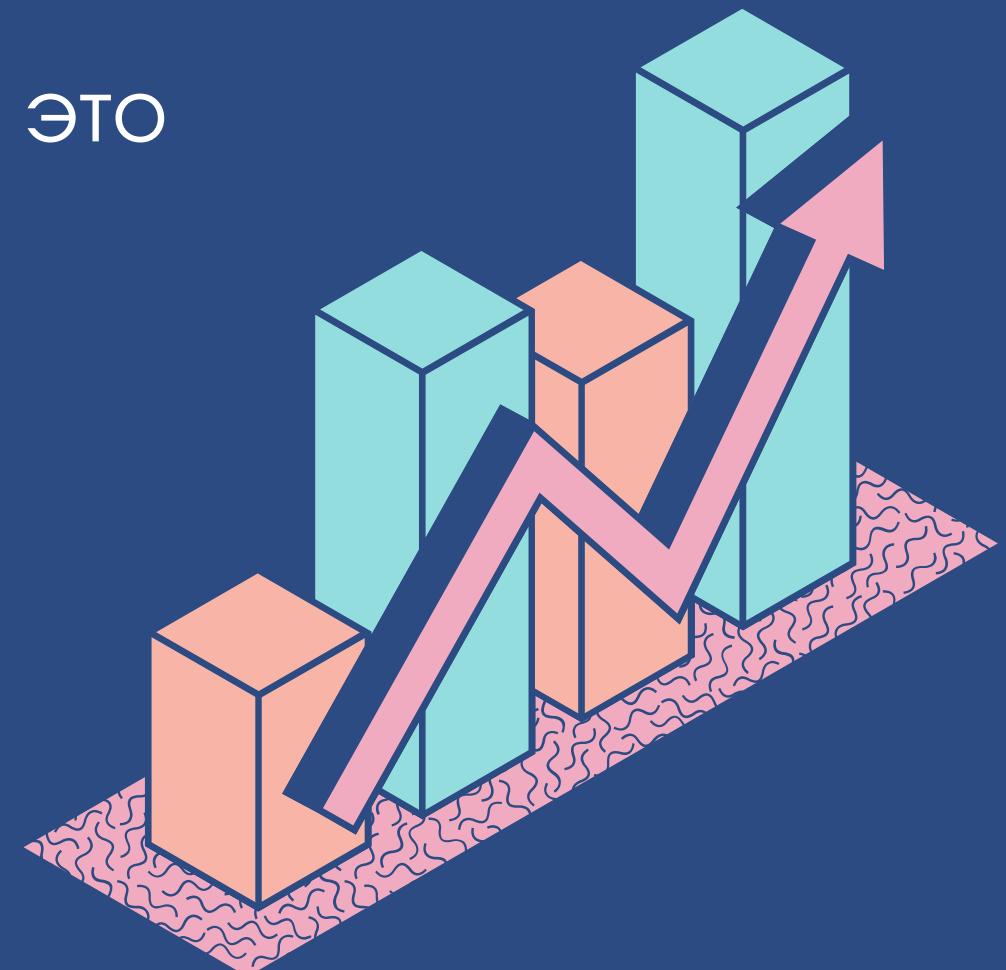
У компонента есть метод для получения данных `fetch`, который вызывается при первом рендрере компонента. Если рендер происходит на сервере, то данные запрашиваются, компонент рендерится, на клиенте происходит гидротация этих данных.

Метод `nuxtServerInit` предназначен для инициализации сторы, вызывается только на сервере, получает данные, складывает их в стору, на клиенте мы получаем предзаполненную стору.

# Инициализация проекта

- Стартуем проект как написано в документации через yarn create nuxt-app.
- Для теста подключим UI-библиотеку BalmUI.
- Для тестов и замеров перфоманса, чтобы было все честно, нам нужно задеплоить наше приложение. Для этого используем heroku.
- Одновременно работают несколько инстансов, отражающие разное состояние оптимизации: ссылки вида

<https://ssr-optimize2.herokuapp.com/>  
<https://ssr-optimize9.herokuapp.com/>



# Первый тест

<https://nuxt-optimize.herokuapp.com/>

ОТКРЫВАЕМ PAGE-SPEED, ВСТАВЛЯЕМ НАШ URL, ЖМЕМ АНАЛИЗИРОВАТЬ, О!!!, 98 ПОПУГАЕВ. НА ЭТОМ МЕСТЕ ИДУТ ХВАЛЫ СОЗДАТЕЛЕЙ VUE И NUXT. А НА МОБИЛКАХ? УПС... 57? ЗА ЧТО? СМОТРИМ НИЖЕ:

- Устраните ресурсы, блокирующие отображение  
`/tailwind.min.css`
- Удалите неиспользуемый код CSS  
`/tailwind.min.css`
- Удалите неиспользуемый код JavaScript 345kiB
- Сократите время до получения первого байта от сервера 1180ms

# Разбираемся

Открываем devtools -> network

Чтобы было понятней, что там грузится, допишем в конфиг:

```
// nuxt.config.js

build: {
  filenames: {
    chunk: () => '[name].[id].[contenthash].js'
  }
}
```

Подключение tailwind

```
<link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.1.2/dist/tailwind.min.css"
rel="stylesheet">
```

Выкидываем, подключаем и библиотеку *balm-ui*

# Вторая попытка

Коммитим, дожидаемся деплоя, смотрим:

PageSpeed Insights [HOME](#) [DOCS](#)

<https://ssr-optimize2.herokuapp.com/> АНАЛИЗИРОВАТЬ

[для мобильных](#) [для компьютеров](#)

80

<https://ssr-optimize2.herokuapp.com/>

▲ 0–49 ■ 50–89 ● 90–100 ⓘ

**Данные наблюдений** — В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки этой страницы.

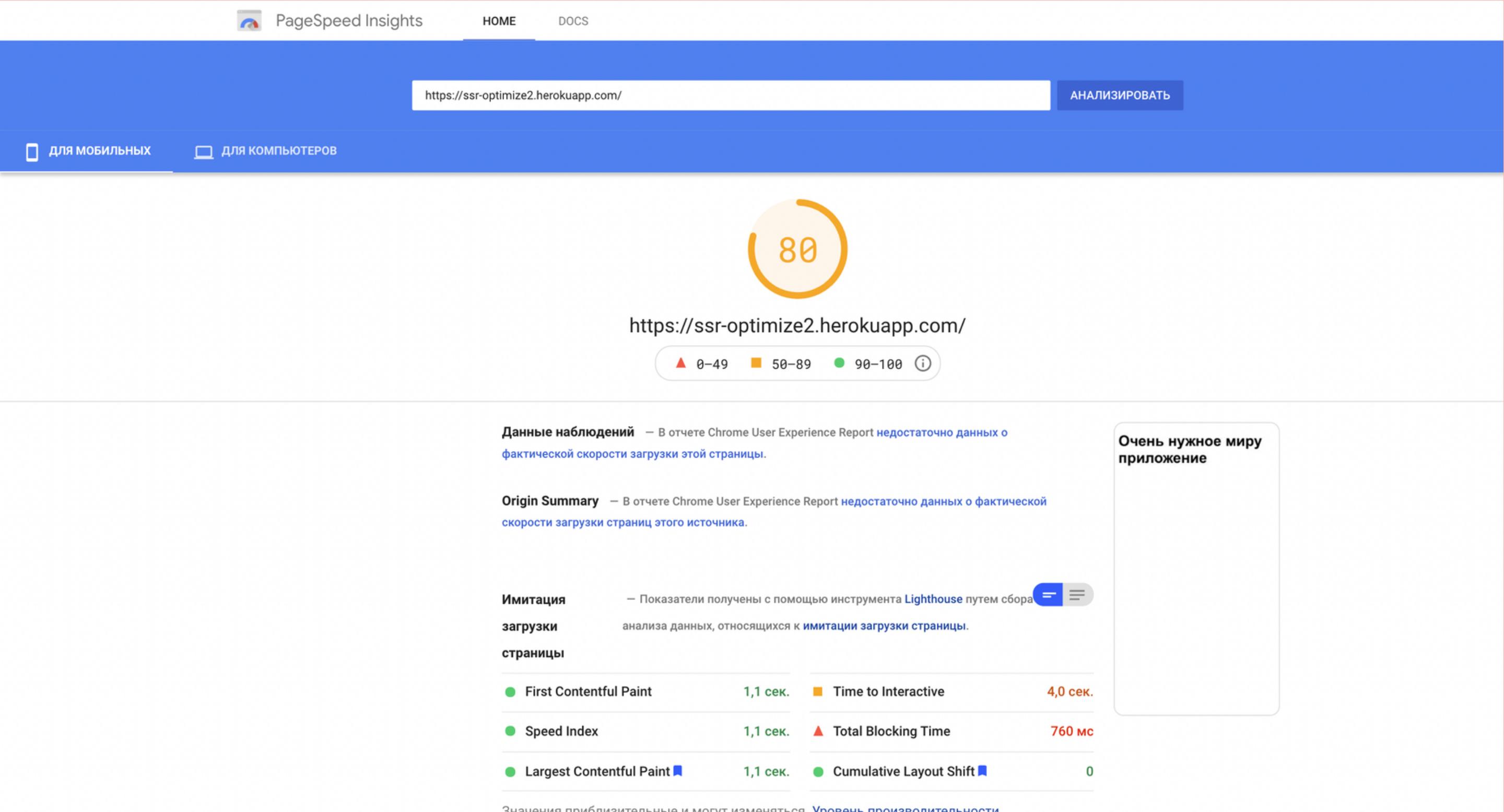
**Origin Summary** — В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки страниц этого источника.

**Имитация загрузки страницы** — Показатели получены с помощью инструмента Lighthouse путем сбора анализа данных, относящихся к имитации загрузки страницы.

|                            |          |                           |          |
|----------------------------|----------|---------------------------|----------|
| ● First Contentful Paint   | 1,1 сек. | ■ Time to Interactive     | 4,0 сек. |
| ● Speed Index              | 1,1 сек. | ▲ Total Blocking Time     | 760 мс   |
| ● Largest Contentful Paint | 1,1 сек. | ● Cumulative Layout Shift | 0        |

Очень нужное миру приложение

Значения приблизительные и могут изменяться. Уровень производительности



# Разбираемся 2

**Сократите время до получения первого байта от сервера 811ms**

Нужно понимать, что происходит при обращении по url нашего приложения. Запрос с клиента попадает в веб-сервер пхт, который рендерит документ. Он делает много работы, поэтому это не быстро.

Скажу только, что лечится это кешированием. Для этого лучше использовать отдельный сервер, например, nginx



# Разбираемся 2

Удалите неиспользуемый код JavaScript 345кiВ

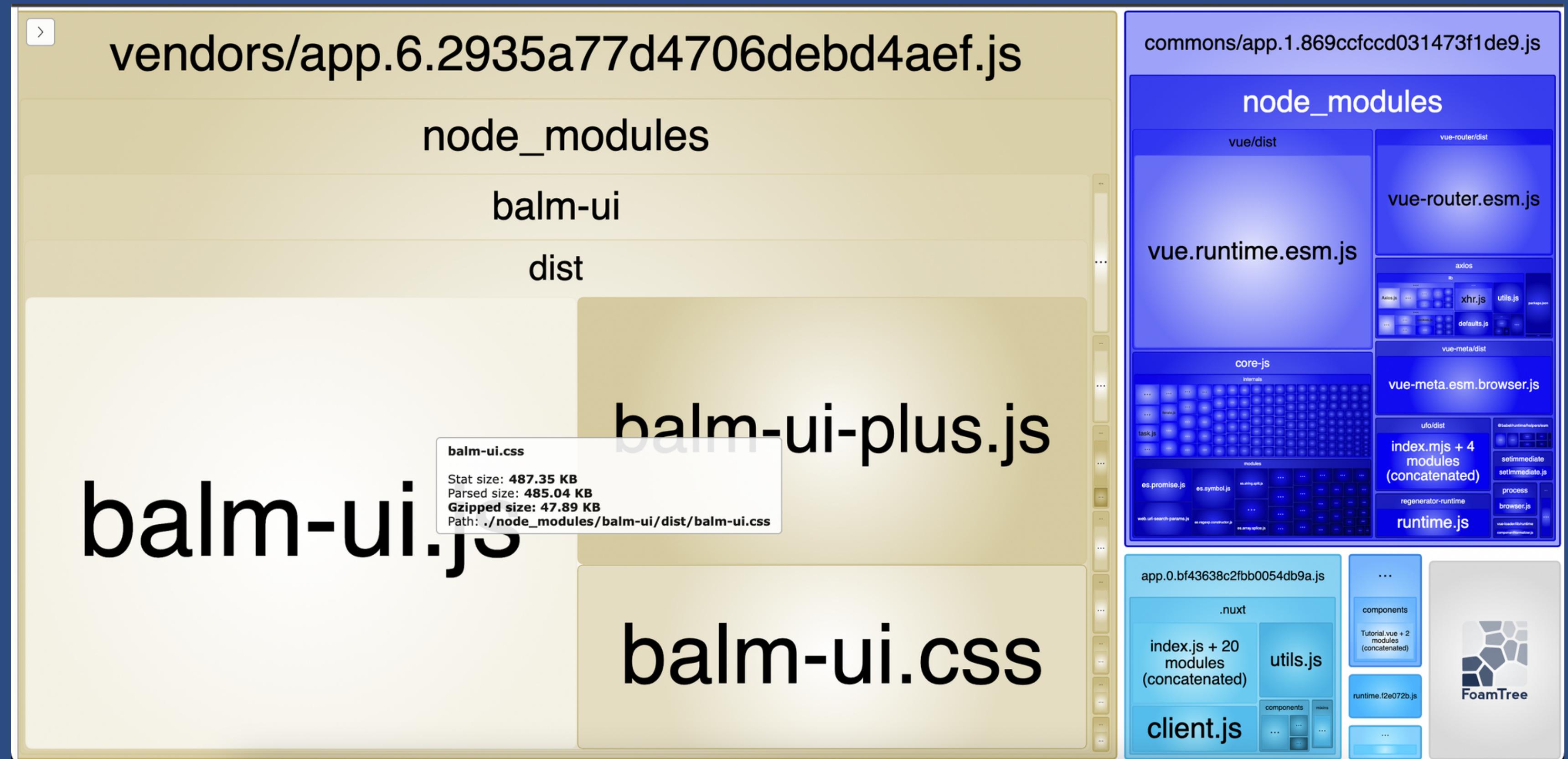
Фреймворк (сам nuxt и vue) вносят какой-то оверхед. А мы его не используем. Но 345 кiВ ?!

В nuxt есть замечательный инструмент для анализа бандла, основанный на webpack-bundle-analyzer:

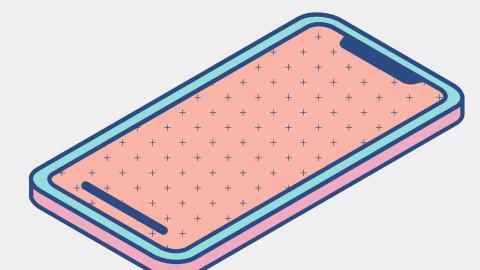
```
нагн nuxt build -а
```



## BOT ОНО: BalmUI

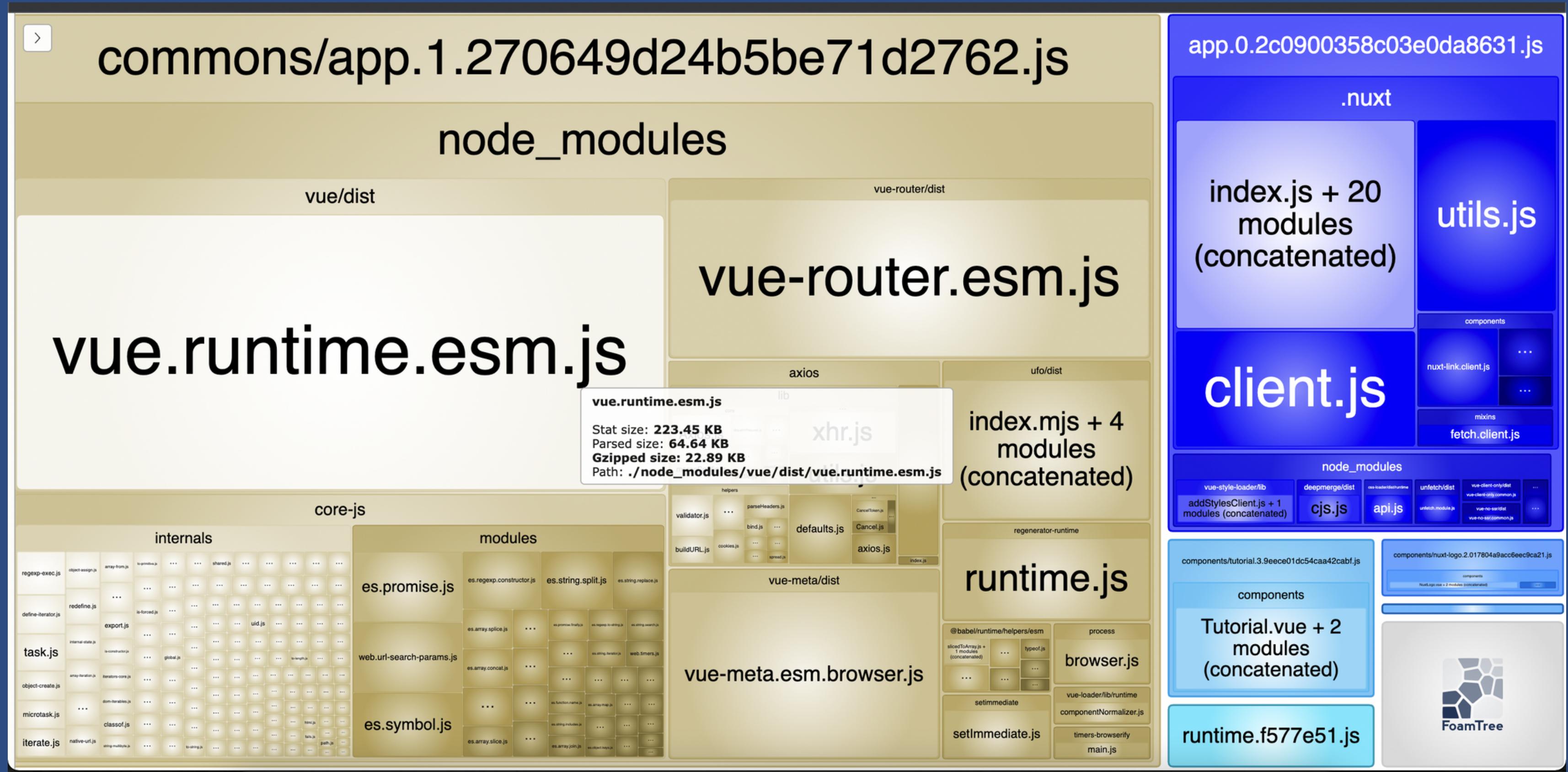


# UI библиотеки



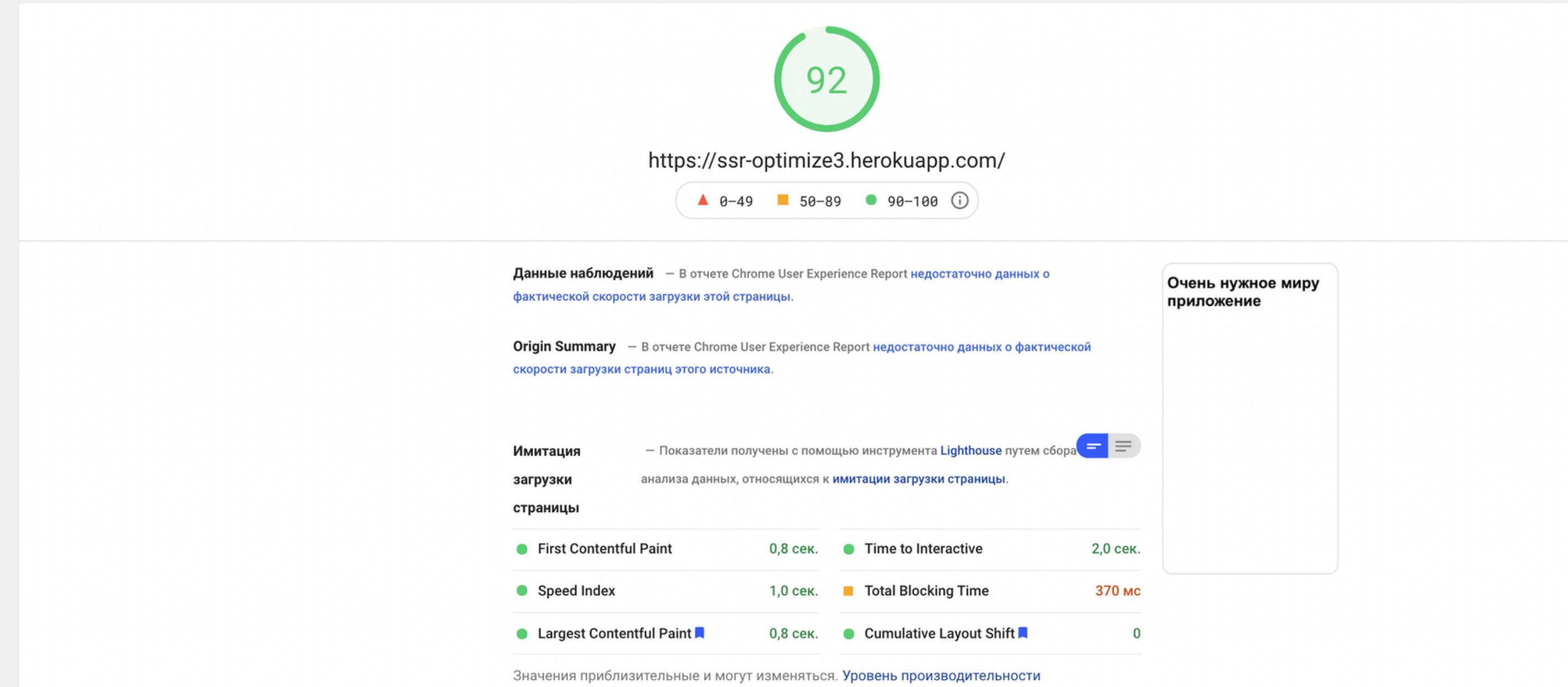
- **А можно ли не подключать библиотеку глобально, а использовать только то, что надо? Так, чтобы в бандл попадали только используемые зависимости?**
- Все зависит от используемой библиотеки, от того как она собирается... Сборщики пока еще не важно умеют в tree-shaking. Далеко не каждая библиотека позволяет это сделать.
- Если на первый вопрос ответ отрицательный, то есть второй вопрос: **готовы ли мы платить за использование этой библиотеки?**
- Тут можно возразить, что на данном этапе код библиотеки на самом деле неиспользуемый, но по мере разработки все больше и больше будет использоваться. Но это не так.

Удаляем подключение библиотеки. Подключим tailwind. Снова запускаем анализ бандла, получаем такую картинку:



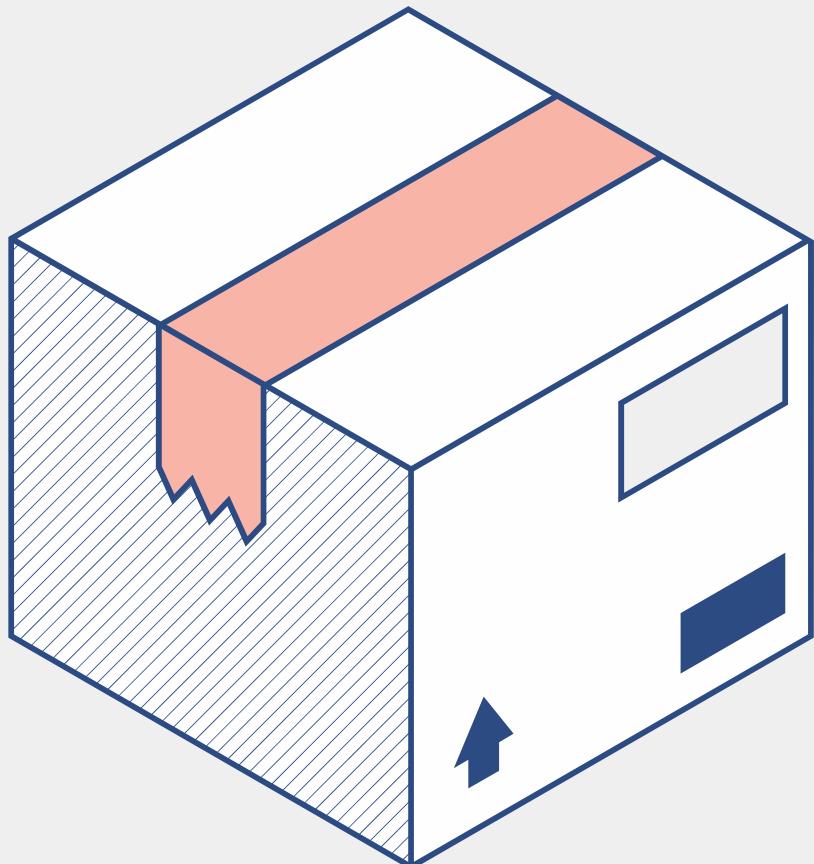
# auto imports

- Уже лучше, но откуда тут Tutorial.vue? Мы же его не используем. Дело в автоИмпорте компонентов. Убираем это в конфиге  
`components: false,`
- Деплоим и тестируем еще раз:

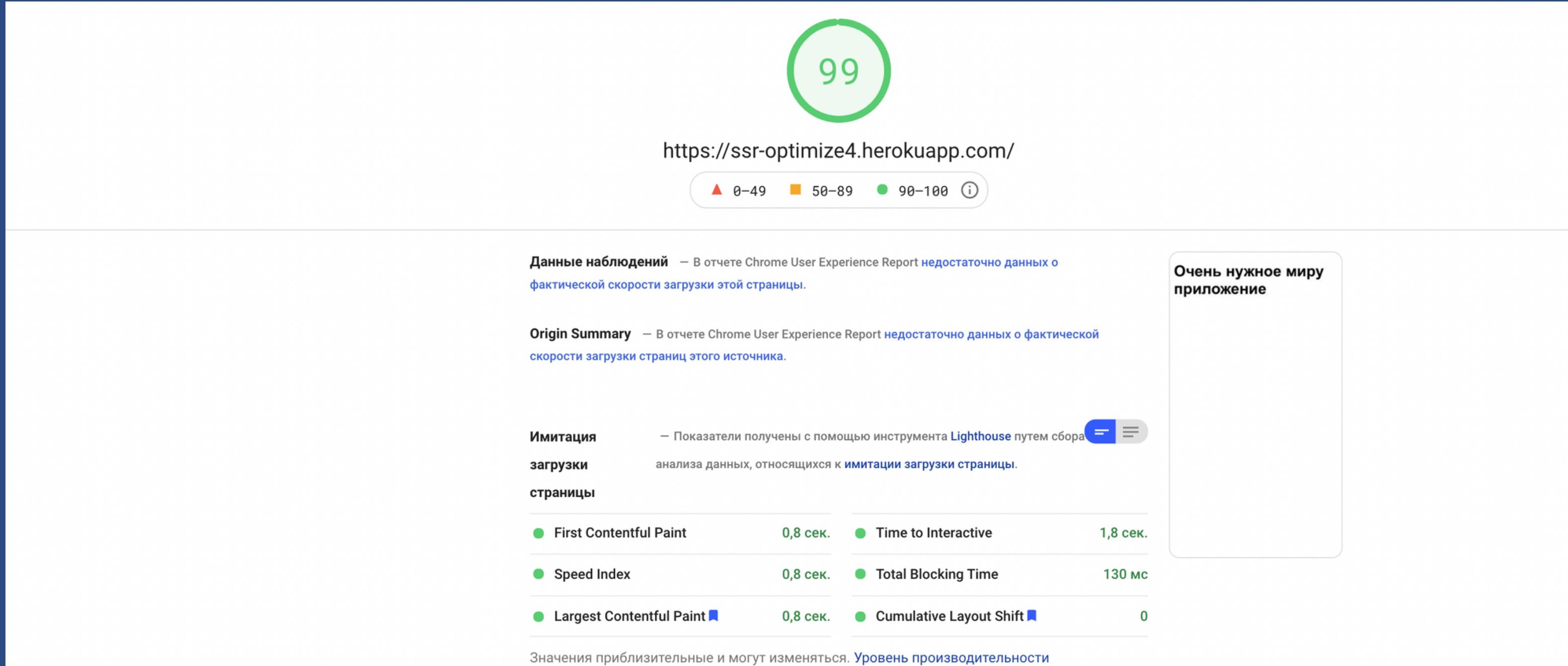


# modern mode

- Вроде 92 это не плохо, но не забываем, что у нас страница на которой только заголовок. **Удалите неиспользуемый код JavaScript 33 KiB**
- Тут подходим к неочевидному моменту. А именно поддержка старых браузеров. Nuxt поддерживает браузеры начиная с IE9. Соответственно код изобилует различными полифилами. Но, все не так плохо. В Nuxt есть опция `modern`. Включаем эту волшебную штуку в конфиге



`modern: true`



# Итоги инициализации

- глобальные зависимости зло
- не забываем про modern
- выключаем auto-import

# Первый запуск

тестовый стенд написан и задиплоен, смотрим что нам показывает page-speed:

https://ssr-optimize5.herokuapp.com/ АНАЛИЗИРОВАТЬ

для МОБИЛЬНЫХ    для КОМПЬЮТЕРОВ

48

https://ssr-optimize5.herokuapp.com/ ▲ 0–49 ■ 50–89 ● 90–100 ⓘ

Данные наблюдений – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки этой страницы.

Origin Summary – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки страниц этого источника.

Имитация загрузки страницы – Показатели получены с помощью инструмента Lighthouse путем сбора анализа данных, относящихся к имитации загрузки страницы.

|                            |          |                           |          |
|----------------------------|----------|---------------------------|----------|
| ● First Contentful Paint   | 1,2 сек. | ● Time to Interactive     | 3,1 сек. |
| ● Speed Index              | 1,8 сек. | ▲ Total Blocking Time     | 1 150 мс |
| ▲ Largest Contentful Paint | 4,5 сек. | ▲ Cumulative Layout Shift | 0,489    |

Значения приблизительные и могут изменяться. Уровень производительности

Показать аудиты All FCP LCP TBT CLS

**Оптимизация** – Эти рекомендации могут помочь вам ускорить загрузку страницы. Они не влияют на показатель производительности [напрямую](#).

Возможности

Приблизительная экономия

|   |  |
|---|--|
| ▲ Используйте современные форматы изображений |  15,3 s   |
| ▲ Настройте эффективную кодировку изображений |  13,2 s   |
| ▲ Настройте подходящий размер изображений     |  9 s    |
| ■ Удалите неиспользуемый код JavaScript       |  0,15 s |

- все картинки переведем в webp;
- отресайзим изображения для разных устройств (у нас один размер 375px);
- зададим размер верхнему изображению (что бы не было смещения при загрузке изображения);
- поставим атрибут `loading="lazy"` для все картинок

https://ssr-optimize6.herokuapp.com/ АНАЛИЗИРОВАТЬ

МБИЛЬНЫХ ДЛЯ КОМПЬЮТЕРОВ

78

https://ssr-optimize6.herokuapp.com/ ▲ 0-49 ■ 50-89 ● 90-100 ⓘ

Данные наблюдений – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки этой страницы.

Origin Summary – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки страниц этого источника.

Имитация загрузки страницы – Показатели получены с помощью инструмента Lighthouse путем сбора анализа данных, относящихся к имитации загрузки страницы.

|                            |          |                           |          |
|----------------------------|----------|---------------------------|----------|
| ● First Contentful Paint   | 1,2 сек. | ● Time to Interactive     | 3,0 сек. |
| ● Speed Index              | 1,6 сек. | ▲ Total Blocking Time     | 690 мс   |
| ■ Largest Contentful Paint | 3,0 сек. | ● Cumulative Layout Shift | 0        |

Значения приблизительные и могут изменяться. Уровень производительности

Основная наша проблема кроется в метрике **Total Blocking Time 690 мс**

# Dynamic imports

Для начала обратимся к такой полезной фиче вебпака, как разделение кода и волшебные комментарии. В компоненте страницы у нас есть такие строчки:

```
import CardList1 from "~/components/CardList/CardList1";
```

Здесь мы импортируем компонент для дальнейшего использования. Вынесем его в отдельный чанк:

```
const CardList1 = () => import(/* webpackChunkName: "CardList1" */  
  '~/components/CardList/CardList1.vue')
```

Запустив приложение и открыв devtools, увидим, что мы грузим на клиент файл

```
CardList1.1.f29585b.modern.js.
```

# Dynamic imports

Заглянем в него.

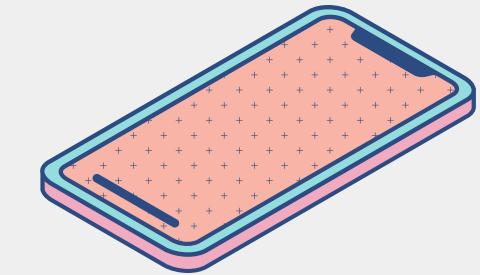
Чанк содержит код нашего компонента

CardList1

А что если он нам не нужен?

- Есть компоненты, которые достаточно отрендерить один раз в виде html и больше ни когда не трогать. В нашем приложении все компоненты такие.
- Для подавляющего количества компонентов верно следующее: если в документе есть разметка компонента, то код для него нам нужен не сразу (либо не нужен вовсе, п.1), а при срабатывании различных триггеров.

# LazyHydration

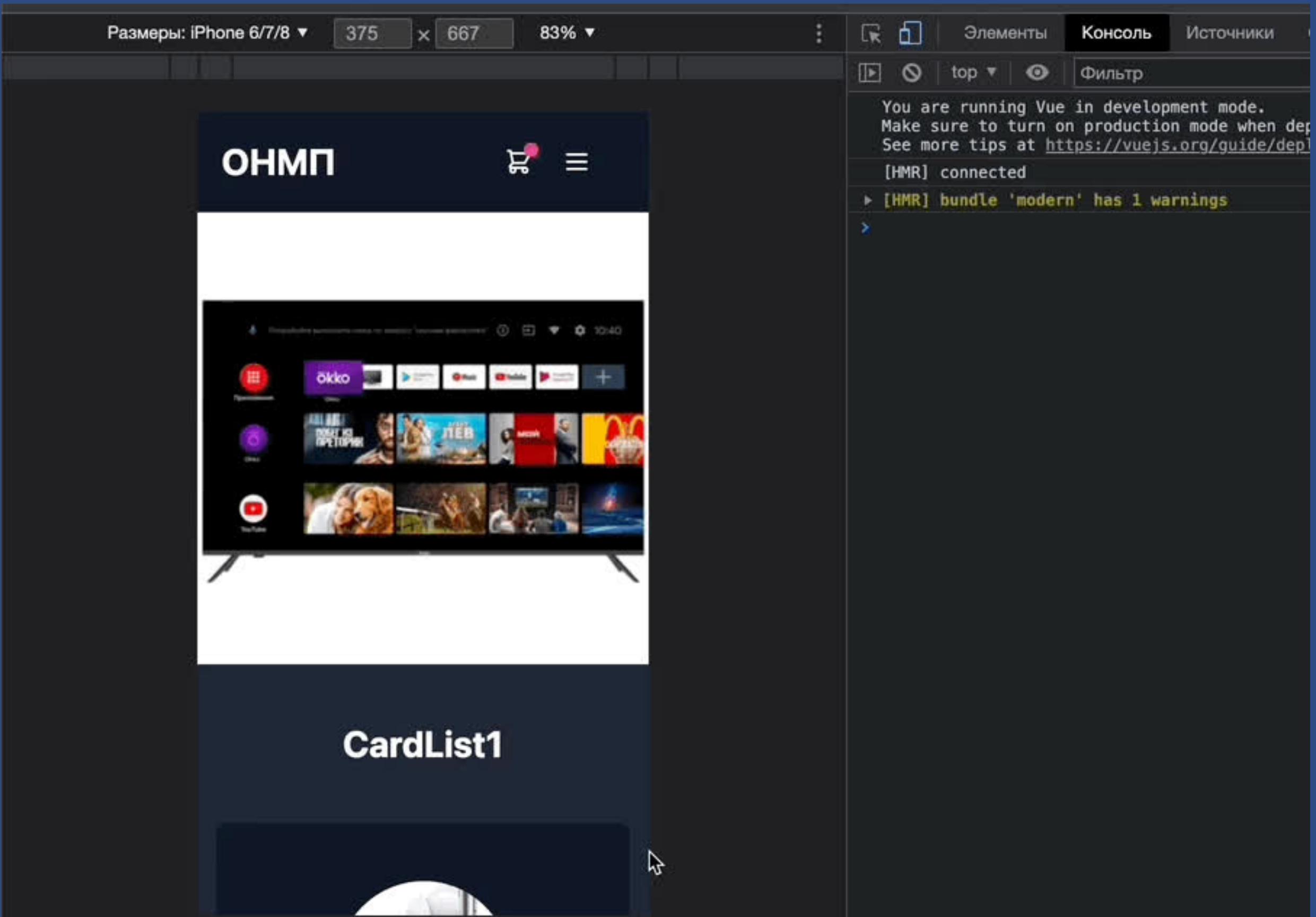


- Тут нам на помощь спешит недооцененная и изящная библиотека vue-lazy-hydration. Она позволяет отложить гидратацию компонента.
- Библиотека предоставляет компоненты, в которые обворачивается целевой компонент

```
<LazyHydrate when-visible>
  <CardList10 :products-group="2"/>
</LazyHydrate>
```

- Вставим в хук mounted компонента CardList console.log, обернем все секции на странице в LazyHydrate, первым двум поставим атрибут never, что говорит LazyHydrate не гидратировать компонент никогда, остальным поставим атрибут when-visible.

запускаем проект,  
скролим вниз и видим,  
что наши сообщения  
появляются не сразу,  
а по мере скrolа  
страницы, начиная с  
компонента CardList3.



```
<template>
  <CardList10
    :products-group="productsGroup"/>
</template>

<script>
const CardList10 = () =>
  import(/* webpackChunkName: "CardList10" */ '~/components/CardList/CardList10.vue')

export default {
  name: "LazyCardList10",
  components: {CardList10},
  props: {
    productsGroup: {
      type: Number,
      required: true
    },
  },
}
</script>
```

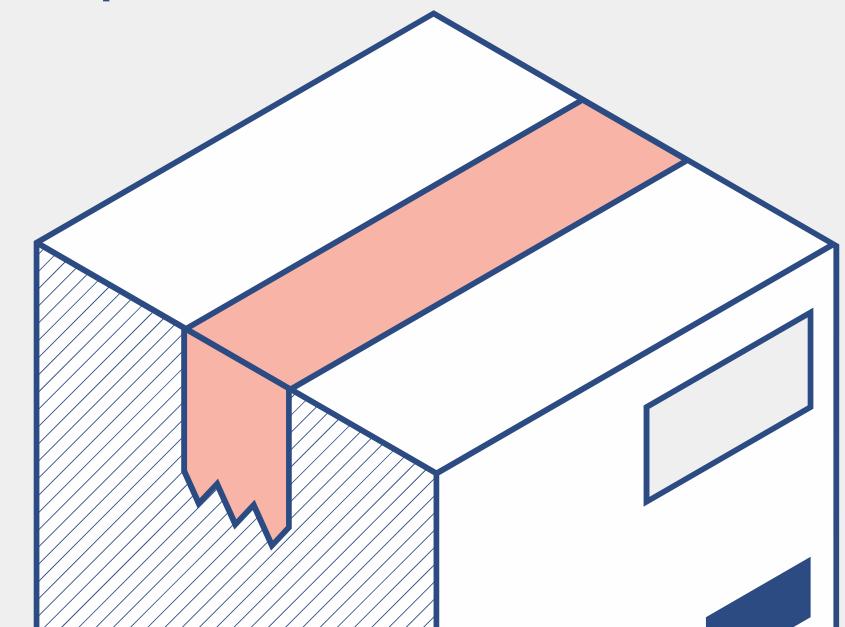
Осталась маленькая проблема: чанк для компонента все еще грузится сразу, дело в том что код компонента **index** исполняется сразу, и импорт дочерних компонентов происходит в момент рендера. Добавим обертку над целевым компонентом:

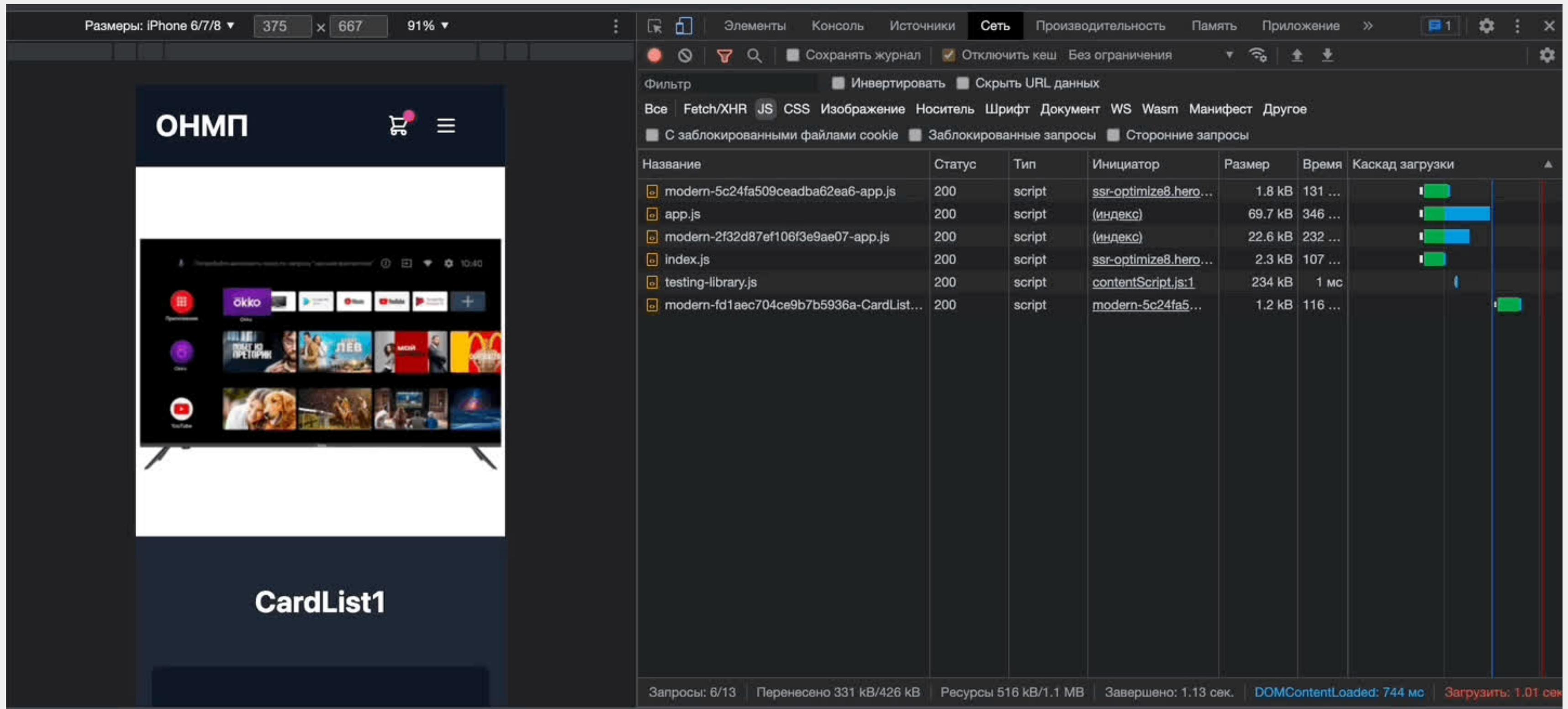
- Все равно грузится... беда! Происходит это из-за того, что ссылки на динамические чанки вставляются в документ:

```
<link rel="preload" href="..." as="script">
```

- На это ишью, до сих пор в обсуждении. К счастью, там же есть решение. Несколько костыльное, но рабочее.
- Теперь загрузка компонентов CardList происходит в момент маунта компонента LazyCardList, который в свою очередь маунтится когда ему "разрешит" LazyHydration. Ставим всем компонентам на странице

```
LazyHydrate when-visible
```





91

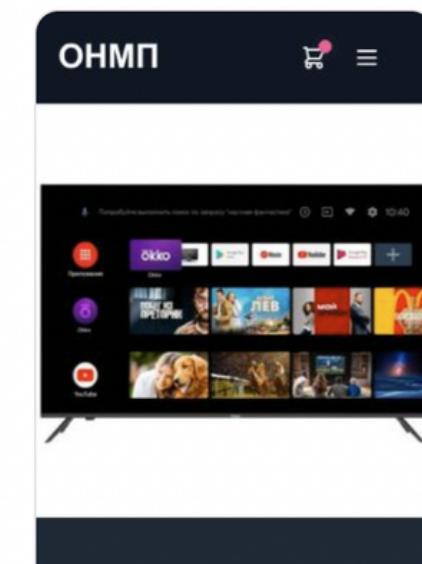
<https://nuxt-optimize.herokuapp.com/>

▲ 0–49 ■ 50–89 ● 90–100 ⓘ

**Данные наблюдений** – В отчете Chrome User Experience Report [недостаточно данных о фактической скорости загрузки этой страницы](#).

**Origin Summary** – В отчете Chrome User Experience Report [недостаточно данных о фактической скорости загрузки страниц этого источника](#).

**Имитация загрузки страницы** – Показатели получены с помощью инструмента Lighthouse путем сбора анализа данных, относящихся к [имитации загрузки страницы](#).



● First Contentful Paint

1,2 сек.

● Time to Interactive

2,4 сек.

● Speed Index

1,5 сек.

● Total Blocking Time

380 мс

● Largest Contentful Paint

1,4 сек.

● Cumulative Layout Shift

0

- В react для ленивой гидратации есть react-lazy-hydration. Реализовал ленивую загрузку чанков в приложении на базе next.js.
- Столкнулся с такой же проблемой как и в приложении nuxt: чанки гружаются в независимости от гидратации компонента. Решение аналогично: убрать ссылки на чанки из документа. Вся магия происходит в файле \_document.js
- Использование простое: при динамическом импорте добавляем к имени чанка DYNAMIC\_IMPORT

```
const Section1 = dynamic(() =>
  import(/*webpackChunkName: "Section1_DYNAMIC_IMPORT"
  */'./Section1').then((module) => module.Section1))
```

### README.md

```
<!doctype html>
<html data-n-head-ssr lang="en" data-n-head="%7B%22lang%22:%7B%22ssr%22:%22en%22%7D%7D">
<head>
  <title>ssr-optimize</title>
  <meta data-n-head="ssr" charset="utf-8">
  <meta data-n-head="ssr" name="viewport" content="width=device-width, initial-scale=1">
  <meta data-n-head="ssr" data-hid="description" name="description" content="">
  <meta data-n-head="ssr" name="format-detection" content="telephone=no">
  <link data-n-head="ssr" rel="icon" type="image/x-icon" href="/favicon.ico">
  <link rel="preload" href="/_nuxt/modern-5c24fa589ceadba62ea6-app.js" as="script">
  <link rel="preload" href="/_nuxt/modern-0132d18578b8258d8277-commons/app.js" as="script">
  <link rel="preload" href="/_nuxt/modern-14af1aa6b1c6157916b2-app.js" as="script">
  <link rel="preload" href="/_nuxt/modern-c537fab69dd96158deb6-pages/index.js" as="script">
  <style
    data-vue-ssr-id="382a115c:0 0b721bb1:0 1af339ee:0">/!* tailwindcss v2.2.17 | MIT License | https://tailwindcss.com
/* cproxia 16      */
/*! modern-normalize v1.1.0 | MIT License | https://github.com/sindresorhus/modern-normalize */</style>
<!--
  Document
  -----
-->
<!--
  Use a better box model (opinionated).
-->

*,::before,::after {
  box-sizing: border-box;
}

<!--
  Use a more readable tab size (opinionated).
-->

html {
  -moz-tab-size: 4;
  -o-tab-size: 4;
  tab-size: 4;
}
```

Структура нашего документа примерно такая:

- строки 16 - 1 084 стили
- строки 1084 - 2 444 разметка
- строки 2444 - 24 660 - данные для гидратации

Внимательно присмотревшись к структуре, можно увидеть строки типа:

```
{  
  layout: "default",  
  data: [{}],  
  fetch: {  
    "CardList1:0": {  
      products: [  
        productId: bb,  
        name: dp,  
        nameTranslit: dq,  
        brandName: as,  
        materialSource: n,  
        // и еще куча полей с вложенными данными  
      ]  
    }  
  }  
}
```

Это результаты наших фетчей. Так же можно найти и наши сторы с предзаполненными данными. Т.е. происходит следующее:

- на сервере компонент запрашивает products
- получает json на 10к строк
- сохраняет это к себе в storу
- данные внедряются на страницу, чтобы потом быть гидратированы

Зачем нам столько данных о товаре, если нам для отображения надо name, category, image ?

Надо что-то с этим делать.

- есть модное решение graphql
- можно научить бекенд принимать фильтр с требуемыми полями  
`?fields="name,image,category.name"`
- Ну и наконец, если один из первых двух вариантов не возможен, то количество гоняемых по сети данных оставим на совести бекендеров, а вот количество данных для гидротации снизим просто добавив мапер

В месте получения данных добавляем вызов мапера:

```
export function mapApiProductsToProducts(productList) {  
  return productList.map(product => ({  
    name: product.name,  
    category: product.category.name,  
    image: product.image  
  }))  
}
```

 PageSpeed Insights

HOME DOCS

□ ДЛЯ МОБИЛЬНЫХ □ ДЛЯ КОМПЬЮТЕРОВ

**96**

<https://nuxt-optimize.herokuapp.com/>

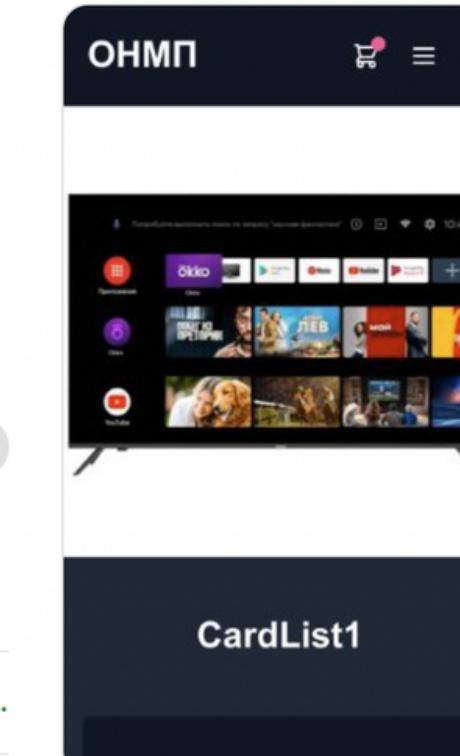
▲ 0–49 ■ 50–89 ● 90–100 ⓘ

**Данные наблюдений** – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки этой страницы.

**Origin Summary** – В отчете Chrome User Experience Report недостаточно данных о фактической скорости загрузки страниц этого источника.

**Имитация загрузки страницы** – Показатели получены с помощью инструмента Lighthouse путем сбора анализа данных, относящихся к имитации загрузки страницы.

|                            |          |                           |          |
|----------------------------|----------|---------------------------|----------|
| ● First Contentful Paint   | 0,9 сек. | ● Time to Interactive     | 2,2 сек. |
| ● Speed Index              | 1,9 сек. | ■ Total Blocking Time     | 220 мс   |
| ● Largest Contentful Paint | 1,6 сек. | ● Cumulative Layout Shift | 0        |

**ОИМП** 



Раз уж мы взялись смотреть на данные для гидратации, давайте посмотрим на них еще раз. **Все еще много** - у нас остался запрос за категориями. С одной стороны, для первого отображения они не нужны, и можно смело перенести этот запрос с сервера на клиент и запрашивать их при клике по бургеру. Но это ухудшит отзывчивость нашего интерфейса. Можно не обращать на это внимания. А можно **разделить этот большой запрос на несколько этапов**:

- на сервере запрашиваем только первый уровень меню
- при открытии меню запрашиваем остальное (или опять только следующий уровень)

PageSpeed Insights [HOME](#) [DOCS](#)

[для МОБИЛЬНЫХ](#) [для КОМПЬЮТЕРОВ](#)



https://ssr-optimize9.herokuapp.com/

▲ 0–49 ■ 50–89 ● 90–100 ⓘ

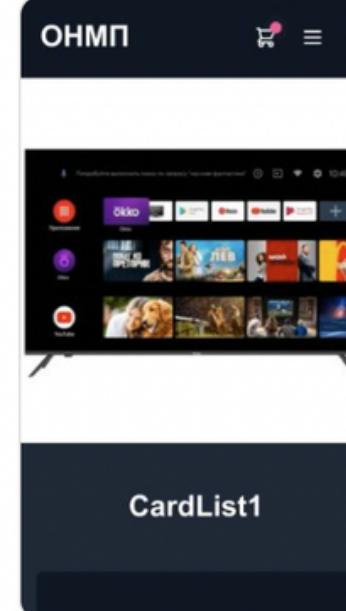
**Данные наблюдений** – В отчете Chrome User Experience Report [недостаточно данных о фактической скорости загрузки этой страницы](#).

**Origin Summary** – В отчете Chrome User Experience Report [недостаточно данных о фактической скорости загрузки страниц этого источника](#).

**Имитация загрузки страницы** – Показатели получены с помощью инструмента [Lighthouse](#) путем сбора анализа данных, относящихся к [имитации загрузки страницы](#).

|                            |          |                           |          |
|----------------------------|----------|---------------------------|----------|
| ● First Contentful Paint   | 0,9 сек. | ● Time to Interactive     | 1,9 сек. |
| ● Speed Index              | 2,3 сек. | ● Total Blocking Time     | 110 мс   |
| ● Largest Contentful Paint | 1,6 сек. | ● Cumulative Layout Shift | 0        |

Значения приблизительные и могут изменяться. Уровень производительности рассчитывается непосредственно на основании этих показателей. [Показать калькулятор](#)



# Заключение

Берегите пользователя. Заботьтесь о его телефончике.  
Не грузите лишнего.

Ну и самое главное: современный фронтенд  
сложный, многоликий, тяжелый... но не забывайте, что  
это все еще просто html, css и js.

Как то так.

