# Введение в оптимизацию производительности.

Производительность веб-сайта в контексте React может относиться к скорости загрузки страниц, быстродействию интерактивных элементов, а также к эффективности обработки данных в приложении. Важность производительности веб-сайта может быть выражена через следующие аспекты:

1. **Улучшенный пользовательский опыт**: Пользователи предпочитают быстрые и отзывчивые сайты. Улучшение производительности может привести к более быстрому отклику на действия пользователя и уменьшению времени ожидания.
2. **Более высокий ранг в поисковой выдаче**: Поисковые системы, такие как Google, используют производительность сайта как один из факторов ранжирования в результатах поиска. Оптимизация производительности может помочь улучшить видимость сайта в поисковой выдаче.
3. **Экономия ресурсов**: Эффективное использование ресурсов, таких как процессор и память, может помочь снизить нагрузку на сервера и устройства пользователей, а также уменьшить расходы на обслуживание инфраструктуры.
4. **Расширяемость**: Оптимизированные приложения легче масштабировать и поддерживать, поскольку они более эффективно используют ресурсы и имеют меньше проблем с производительностью.

В контексте React, оптимизация производительности может включать в себя следующие стратегии:

- **Использование метода `shouldComponentUpdate` или `React.memo` для предотвращения ненужных рендеров**.
- **Применение ленивой загрузки (lazy loading) и разделения кода (code splitting)** для уменьшения времени загрузки.
- **Оптимизация состояния и пропсов компонентов**, чтобы уменьшить количество обновлений DOM.
- **Использование виртуальных списков и окон (windowing)** для эффективного отображения больших списков данных.
- **Профилирование производительности** с помощью инструментов, таких как React DevTools, для выявления и устранения узких мест в производительности.

Осознание и оптимизация производительности являются важными аспектами разработки на React, которые могут привести к созданию более эффективных и приятных для пользователя веб-приложений.

# Разберёмся с каждым из приведенных выше методов оптимизации производительности на примерах.

---

## 1. `shouldComponentUpdate`:

Метод `shouldComponentUpdate` позволяет указать, должен ли компонент обновляться в ответ на изменение состояния или пропсов. Этот метод используется только в классовых компонентах. В место него используем аналог `shouldComponentUpdate` для функциональных компонентов — это `React.memo`.

Пример с `React.memo`:

```
import React, { useState } from "react";

const MyComponent = React.memo(function MyComponent({ value }) {
    console.log("Rendered!");
```

```
    return <div>{value}</div>;
});

function App() {
    const [value, setValue] = useState(0);

    return (
        <div>
            <MyComponent value={value} />
            <button onClick={() => setValue(value + 1)}>Increment</button>
        </div>
    );
}

export default App;
```

В этом примере `React.memo` оборачивает `MyComponent`, предотвращая его повторный рендеринг, если пропсы не изменились. Каждый раз, когда вы нажимаете кнопку, `value` увеличивается, но `MyComponent` рендерится только при первом клике, потому что его пропсы не изменяются с последующими кликами.

## 2. `React.lazy` и `React.Suspense`:

`React.lazy` и `React.Suspense` позволяют загружать компоненты асинхронно, что уменьшает время загрузки приложения.

Пример:

```
import React, { Suspense } from "react";

const LazyComponent = React.lazy(() => import("./LazyComponent"));

function App() {
    return (
        <div>
            <Suspense fallback={<div>Loading...</div>}>
                <LazyComponent />
            </Suspense>
        </div>
    );
}

export default App;
```

Здесь `React.lazy` загружает `LazyComponent` асинхронно. `React.Suspense` отображает запасной контент (`fallback`), пока `LazyComponent` не будет загружен

и готов к рендерингу.

## 3. Lazy Loading в Vite:

Vite предлагает встроенную поддержку lazy loading с помощью динамического `import()`. Это похоже на `React.lazy`, но может быть использовано вне контекста React.

Пример:

```
// В вашем компоненте или файле...
const MyComponent = React.lazy(() => import("./MyComponent"));

// В файле MyComponent.js...
export default function MyComponent() {
    return <div>Hello from MyComponent</div>;
}
```

Этот пример аналогичен примеру с `React.lazy`, но здесь мы используем динамический `import()` напрямую, что поддерживается Vite.

# Метрики производительности

Метрики производительности важны для того, чтобы понять, насколько хорошо работает ваш веб-сайт или приложение, и где есть потенциал для улучшения. Вот несколько ключевых метрик производительности:

1. **Время загрузки страницы (Page Load Time, PLT)**:
    - Это время, необходимое для полной загрузки страницы от момента запроса пользователя до момента, когда страница полностью отрисована и готова к использованию.
    - Оптимизации, направленные на уменьшение времени загрузки, включают минимизацию размера ресурсов, оптимизацию изображений и применение техник, таких как асинхронная загрузка и кэширование.
2. **Кадры в секунду (Frames Per Second, FPS)**:

- FPS измеряет, сколько кадров отрисовывается в секунду. Оптимальное значение — 60 FPS, что обеспечивает плавное визуальное восприятие.
- Низкое значение FPS может указывать на проблемы с производительностью, такие как утечки памяти или интенсивные вычисления, которые блокируют основной поток.

3. **Время отклика на взаимодействие пользователя (Time to Interactive, TTI)**:
   - Это время, необходимое для того, чтобы страница стала полностью интерактивной и отзывчивой на взаимодействие пользователя.
   - Чем быстрее страница становится интерактивной, тем быстрее пользователь может начать взаимодействовать с ней.

4. **Первая контентная отрисовка (First Contentful Paint, FCP)**:
   - Это время от начала загрузки до момента, когда первый бит контента (текст, изображения, и т.д.) отрисовывается на экране.
   - Быстрая первая контентная отрисовка улучшает восприятие производительности пользователями.

5. **Первая значимая отрисовка (First Meaningful Paint, FMP)**:
   - Это время от начала загрузки до момента, когда основной контент страницы отрисован и видим пользователем.
   - Эта метрика помогает понять, насколько быстро пользователи получают полезный контент.

6. **Кумулятивное смещение макета (Cumulative Layout Shift, CLS)**:
   - Это мера нестабильности макета во время загрузки страницы. Она измеряет, насколько неожиданно элементы смещаются в процессе загрузки.
   - Низкое значение CLS указывает на стабильность макета, что улучшает восприятие производительности пользователями.

7. **Время задержки ввода (Input Delay, ID)**:
   - Это время между моментом, когда пользователь взаимодействует с интерфейсом (например, нажимает на кнопку), и моментом, когда браузер может начать обрабатывать это взаимодействие.

- Более короткое время задержки ввода улучшает отзывчивость приложения и улучшает пользовательский опыт.

Эти метрики могут быть измерены с помощью инструментов, таких как Google Lighthouse, WebPageTest или с помощью встроенных инструментов разработчика в вашем браузере. Они помогают разработчикам выявлять и устранять узкие места в производительности, чтобы улучшить пользовательский опыт и общее качество веб-сайтов и приложений.

# Анализ производительности.

## Вкладка "Network":

1. **Открытие DevTools и вкладки Network**:
   - Откройте Chrome, затем нажмите `Ctrl+Shift+I` (или `Cmd+Opt+I` на Mac) для открытия DevTools.
   - Перейдите на вкладку "Network".
2. **Анализ загрузки ресурсов**:
   - Обновите страницу (`F5` или `Cmd+R` на Mac), чтобы увидеть все загружаемые ресурсы.
   - Каждый ресурс представлен строкой в таблице, где указаны имя ресурса, его размер, время загрузки и другие параметры.
3. **Временная шкала (Waterfall)**:
   - Временная шкала показывает, когда каждый ресурс начал загружаться и сколько времени заняла его загрузка.
   - С помощью временной шкалы можно выявить ресурсы, которые занимают много времени для загрузки, и оптимизировать их.

## Вкладка "Performance":

1. **Открытие вкладки Performance**:
   - В DevTools перейдите на вкладку "Performance".

2. **Запуск анализа производительности**:
    - Нажмите на кнопку "Record" (круглая красная кнопка) или `Ctrl+E` (или `Cmd+E` на Mac) для начала записи профилирования производительности.
    - Обновите страницу или взаимодействуйте с ней, чтобы собрать данные производительности.
    - Нажмите на кнопку "Stop" для остановки записи.
3. **Анализ данных производительности**:
    - В разделе "Summary" можно увидеть общее время загрузки, FPS и использование ресурсов.
    - Временные диаграммы показывают, что происходило во время загрузки страницы, включая парсинг HTML, выполнение JavaScript, отрисовку и т.д.
4. **Идентификация узких мест**:
    - Просмотрите разделы "Main", "Frames", "Interactions" и др., чтобы увидеть, какие части кода или ресурсы вызывают замедление.
    - Используйте инструменты, такие как "Flame Chart", чтобы увидеть, какие функции занимали больше всего времени.

Используя вкладки "Network" и "Performance" в Chrome DevTools, можно получить детальный анализ времени загрузки страницы и идентифицировать области, требующие оптимизации, что в свою очередь поможет улучшить общую производительность вашего веб-сайта или приложения.

# Проблемы производительности.

1. **Ненужные рендеры**:
    - React может повторно рендерить компоненты в ответ на изменения состояния или пропсов, даже если это не влияет на вывод компонента. Это может привести к ненужным рендерам и уменьшению производительности.
2. **Утечки памяти**:

- Утечки памяти могут возникнуть из-за неправильного управления ресурсами, например, неотмененные обещания, таймеры или подписки на события.

3. **Блокировка основного потока**:
   - Долгие задачи JavaScript, такие как интенсивные вычисления или обработка больших массивов данных, могут блокировать основной поток и ухудшить отзывчивость приложения.

4. **Неоптимальное изменение состояния**:
   - Неэффективное управление состоянием или частые обновления состояния могут привести к избыточным рендерам и ухудшению производительности.

5. **Неоптимизированные селекторы или вычисления**:
   - Вычисления в рендере или неоптимизированные селекторы могут привести к избыточным вычислениям и уменьшению производительности.

6. **Неправильное использование ключей**:
   - Неправильное использование ключей в списках может привести к непредсказуемому поведению рендеринга и уменьшению производительности.

7. **Недостаточное разделение кода**:
   - Отсутствие разделения кода может привести к загрузке больших объемов кода, что увеличивает время загрузки страницы.

8. **Неправильное использование или отсутствие мемоизации**:
   - Мемоизация может помочь предотвратить повторные вычисления или рендеры, но неправильное ее использование может привести к проблемам.

9. **Неоптимальное использование сторонних библиотек**:
   - Некоторые сторонние библиотеки могут быть неоптимизированными и снижать производительность вашего приложения.

10. **Неправильное управление ресурсами**:
    - Неудачное управление ресурсами, такими как изображения, сетевые запросы или обработчики событий, может привести к утечкам памяти и другим проблемам производительности.

Оптимизация производительности в React-приложениях часто включает в себя идентификацию и решение этих и других проблем, что помогает создать более отзывчивое и эффективное приложение.