Chapter 09: Tối ưu hóa hiệu năng với Next.js

A. Mục tiêu

Sau bài học này, học viên sẽ:

- Hiểu và sử dụng **next/image** để tối ưu hóa hình ảnh, giảm thời gian tải trang và cải thiện trải nghiệm người dùng.
- Tích hợp và tối ưu hóa font chữ với **next/font**, loại bỏ các yêu cầu mạng không cần thiết và ngăn chặn layout shift.
- Sử dụng **next/dynamic** để lazy-load các component, giảm kích thước JavaScript ban đầu và cải thiện hiệu năng.
- Nắm được cơ chế **code splitting** tự động của Next.js và cách tận dụng nó để tối ưu hóa ứng dụng.
- Biết cách kiểm tra hiệu năng ứng dụng qua DevTools và các công cụ phân tích như Lighthouse.

B. Nội dung lý thuyết

1. Tối ưu hóa hình ảnh với next/image

Hình ảnh là một trong những yếu tố ảnh hưởng lớn đến hiệu năng của trang web. Thẻ `<img>` truyền thống có nhiều hạn chế:

- **Tải ảnh kích thước gốc**: Dù ảnh chỉ hiển thị nhỏ, trình duyệt vẫn tải toàn bộ kích thước gốc.
- **Không có lazy-loading mặc định**: Ảnh được tải ngay cả khi nằm ngoài khung nhìn.
- **Cumulative Layout Shift (CLS)**: Giao diện bị xê dịch khi ảnh tải xong, gây khó chịu cho người dùng.

next/image là giải pháp tích hợp sẵn trong Next.js để giải quyết các vấn đề trên:

- **Tự động thay đổi kích thước**: Phục vụ ảnh với kích thước phù hợp dựa trên viewport của người dùng.
- **Tối ưu hóa định dạng**: Chuyển đổi ảnh sang các định dạng hiện đại như **WebP** hoặc **AVIF** nếu trình duyệt hỗ trợ.
- **Lazy-loading mặc định**: Ảnh chỉ tải khi sắp xuất hiện trong khung nhìn.
- **Chống layout shift**: Tự động giữ chỗ cho ảnh dựa trên kích thước được khai báo.

Cú pháp cơ bản:

- Cho ảnh local:

```
import Image from 'next/image';
import myImage from '../public/images/my-image.jpg';

export default function MyComponent() {
  return (
    <Image
      src={myImage}
      alt="Description"
      width={500}
      height={300}
      placeholder="blur" // Hiển thị hiệu ứng mờ khi ảnh đang tải
    />
  );
}
```

- Cho ảnh từ URL (ví dụ từ CMS):

```
import Image from 'next/image';

export default function MyComponent() {
  return (
    <div style={{ position: 'relative', width: '100%', height: '300px' }
      <Image
        src="https://example.com/image.jpg"
        alt="Description"
```

```
        fill
        style={{ objectFit: 'cover' }}
      />
    </div>
  );
}
```

Thuộc tính quan trọng:

- `priority` : Tải ảnh ngay lập tức (dùng cho ảnh above-the-fold như banner).
- `placeholder="blur"` : Hiển thị ảnh mờ trong khi tải (chỉ hoạt động với ảnh local).
- `quality` : Điều chỉnh chất lượng ảnh (mặc định: 75).
- `sizes` : Chỉ định kích thước ảnh dựa trên viewport (ví dụ: `(max-width: 768px) 100vw, 50vw`).

Lưu ý:

- Ảnh local phải đặt trong thư mục `public` .
- Khi dùng ảnh từ URL, cần cấu hình domain trong `next.config.js` :

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  images: {
    domains: ['example.com'],
  },
};
module.exports = nextConfig;
```

---

2. Tối ưu hóa Font với next/font

Việc sử dụng font tùy chỉnh (như Google Fonts) qua thẻ `<link>` có thể gây ra:

- **Yêu cầu mạng bổ sung**: Trình duyệt phải tải font từ `fonts.googleapis.com`.
- **Layout shift**: Giao diện bị xê dịch khi font tùy chỉnh được áp dụng.
- **Chặn render**: Font tải chậm có thể làm chậm thời gian hiển thị nội dung.

next/font giải quyết các vấn đề này bằng cách:

- **Self-host font**: Font được tải về server tại thời điểm build và phục vụ cùng các tài nguyên khác.
- **Tối ưu hóa hiệu năng**: Giảm số lượng yêu cầu mạng, cải thiện thời gian tải.
- **Chống layout shift**: Cung cấp thông số để trình duyệt giữ chỗ cho văn bản trước khi font tải.

Cú pháp:

```
import { Roboto } from 'next/font/google';

const roboto = Roboto({
  subsets: ['latin'],
  weight: ['400', '700'],
  style: ['normal', 'italic'],
});

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={roboto.className}>
      <Component {...pageProps} />
    </main>
  );
}
```

Thuộc tính quan trọng:

- `subsets`: Chỉ định tập hợp ký tự cần tải (ví dụ: `latin`, `cyrillic`).
- `weight`: Trọng lượng font (ví dụ: `400`, `700`).
- `style`: Kiểu font (ví dụ: `normal`, `italic`).

- `variable` : Tạo CSS variable để sử dụng font linh hoạt (ví dụ: `--font-roboto`).

Lưu ý:

- Chỉ import font trong `pages/_app.js` để áp dụng toàn cục.
- Kiểm tra DevTools (Network tab) để xác nhận không có yêu cầu đến `fonts.googleapis.com` .

---

3. Lazy Loading Components với next/dynamic

Next.js tự động thực hiện **code splitting** theo trang, nghĩa là mỗi trang chỉ tải JavaScript cần thiết. Tuy nhiên, trong một trang, có thể có các component nặng (ví dụ: thư viện biểu đồ, modal phức tạp) không cần tải ngay từ đầu.

next/dynamic cho phép lazy-load các component, tách chúng thành file JavaScript riêng và chỉ tải khi cần thiết.

Cú pháp:

```
import dynamic from 'next/dynamic';

const HeavyComponent = dynamic(() => import('../components/HeavyComponent'),
  ssr: false, // Tắt Server-Side Rendering nếu component chỉ chạy ở client
  loading: () => <p>Đang tải...</p>, // Hiển thị placeholder khi component đ
});

export default function MyPage() {
  return (
    <div>
      <h1>Trang chính</h1>
      <HeavyComponent />
    </div>
  );
}
```

Ưu điểm:

- Giảm kích thước JavaScript ban đầu, cải thiện thời gian tải trang.
- Chỉ tải component khi cần (ví dụ: khi người dùng tương tác).
- Hỗ trợ placeholder để cải thiện trải nghiệm người dùng.

Lưu ý:

- Sử dụng `ssr: false` cho các component phụ thuộc vào `window` hoặc `document`.
- Đảm bảo placeholder có kích thước tương tự component để tránh layout shift.

---

4. Code Splitting tự động trong Next.js

Next.js tự động chia nhỏ mã JavaScript theo các tiêu chí:

- **Theo trang**: Mỗi file trong thư mục `pages` được tách thành một bundle riêng.
- **Theo component động**: Các component dùng `next/dynamic` được tách thành bundle riêng.
- **Theo thư viện bên thứ ba**: Các thư viện lớn (như `lodash`, `chart.js`) được tách nếu dùng `next/dynamic`.

Lợi ích:

- Giảm thời gian tải trang đầu tiên (First Contentful Paint).
- Chỉ tải mã cần thiết cho trang hiện tại.
- Cải thiện hiệu năng trên thiết bị di động hoặc mạng chậm.

Kiểm tra code splitting:

- Mở DevTools → Network tab → Lọc các file `.js`.
- Quan sát các file JavaScript được tải khi chuyển trang hoặc tương tác.

C. Bài tập thực hành

Bài 1: Thay thế thẻ `<img>` bằng `next/image`

1. Trong dự án Blog, tìm tất cả các thẻ `<img>` (ví dụ: ảnh tác giả, ảnh bài viết).
2. Thay thế bằng component `Image`:

```
import Image from 'next/image';

export default function AuthorCard() {
  return (
    <div>
      <Image
        src="/images/author.jpg"
        alt="Author"
        width={100}
        height={100}
        className="rounded-full"
      />
      <h3>John Doe</h3>
    </div>
  );
}
```

3. Đảm bảo ảnh được đặt trong thư mục `public/images`.
4. Kiểm tra xem ảnh có tải đúng và không gây layout shift.

Bài 2: Quan sát hiệu năng của `next/image`

1. Mở DevTools → Network tab → Lọc các file hình ảnh.
2. Thêm thuộc tính `priority` cho ảnh chính (ví dụ: banner ở đầu trang):

```
<Image
  src="/images/banner.jpg"
```

```
  alt="Banner"
  width={1200}
  height={400}
  priority
/>
```

3. Quan sát thứ tự tải ảnh: Ảnh có `priority` được tải trước các ảnh khác.
4. Thử thêm `placeholder="blur"` và kiểm tra hiệu ứng khi ảnh tải.

Bài 3: Tích hợp font với `next/font`

1. Trong `pages/_app.js`, tích hợp font `Roboto`:

```
import { Roboto } from 'next/font/google';

const roboto = Roboto({
  subsets: ['latin'],
  weight: ['400', '700'],
});

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={roboto.className}>
      <Component {...pageProps} />
    </main>
  );
}
```

2. Thêm style toàn cục trong `styles/globals.css`:

```
body {
  margin: 0;
  padding: 0;
  background-color: #f4f4f4;
}
```

3. Mở DevTools → Network tab → Xác nhận không có yêu cầu đến `fonts.googleapis.com`.
4. Kiểm tra giao diện để đảm bảo font được áp dụng.

Bài 4: Tạo component biểu đồ với `react-chartjs-2`

1. Cài đặt thư viện:

```
npm install react-chartjs-2 chart.js
```

2. Tạo component `MyChart.js`:

```
import { Bar } from 'react-chartjs-2';
import { Chart as ChartJS, CategoryScale, LinearScale, BarElement, Title

ChartJS.register(CategoryScale, LinearScale, BarElement, Title, Tooltip,

export default function MyChart() {
  const data = {
    labels: ['January', 'February', 'March', 'April'],
    datasets: [
      {
        label: 'Sales',
        data: [65, 59, 80, 81],
        backgroundColor: 'rgba(75, 192, 192, 0.2)',
        borderColor: 'rgba(75, 192, 192, 1)',
        borderWidth: 1,
      },
    ],
  };

  return (
    <div>
      <h2>Sales Report</h2>
      <Bar data={data} />
    </div>
```

```
  );
}
```

Bài 5: Lazy-load component biểu đồ với `next/dynamic`

1. Trong `pages/index.js`, sử dụng `next/dynamic` để lazy-load `MyChart`:

```
import { useState } from 'react';
import dynamic from 'next/dynamic';

const MyChart = dynamic(() => import('../components/MyChart'), {
  ssr: false,
  loading: () => <p>Đang tải biểu đồ...</p>,
});

export default function Home() {
  const [showChart, setShowChart] = useState(false);

  return (
    <div className="min-h-screen flex flex-col items-center justify-cent
      <button
        onClick={() => setShowChart(true)}
        className="bg-blue-500 text-white px-4 py-2 rounded"
      >
        Hiển thị báo cáo
      </button>
      {showChart && <MyChart />}
    </div>
  );
}
```

2. Mở DevTools → Network tab → Nhấn nút “Hiển thị báo cáo”.
3. Quan sát file JavaScript của `MyChart` chỉ được tải sau khi nhấn nút.

D. Bổ sung: Kiểm tra hiệu năng với Lighthouse

Để đánh giá hiệu năng sau khi áp dụng các tối ưu hóa, sử dụng công cụ **Lighthouse** trong DevTools:

1. Mở DevTools → Tab Lighthouse.
2. Chọn chế độ “Mobile” và “Performance”.
3. Chạy phân tích và kiểm tra các chỉ số:
    - **First Contentful Paint (FCP)**: Thời gian hiển thị nội dung đầu tiên.
    - **Largest Contentful Paint (LCP)**: Thời gian tải nội dung chính.
    - **Cumulative Layout Shift (CLS)**: Mức độ xê dịch giao diện.
4. So sánh kết quả trước và sau khi áp dụng `next/image`, `next/font`, và `next/dynamic`.

Mẹo cải thiện điểm Lighthouse:

- Sử dụng `priority` cho các tài nguyên quan trọng (như ảnh banner).
- Giảm số lượng thư viện bên thứ ba.
- Tối ưu hóa kích thước font bằng cách chỉ tải các `subsets` và `weight` cần thiết.

E. So sánh các phương pháp tối ưu hóa

Phương pháp	Ưu điểm	Nhược điểm
next/image	Giảm kích thước ảnh, lazy-load tự động, chống layout shift.	Cần cấu hình domain cho ảnh từ URL, yêu cầu khai báo kích thước.
next/font	Self-host font, loại bỏ yêu cầu mạng, chống layout shift.	Cần chọn subsets và weight cẩn thận để tránh tải font thừa.
next/dynamic	Giảm kích thước JavaScript ban đầu, chỉ tải khi cần.	Có thể gây trễ nhẹ khi tải component lần đầu, cần placeholder hợp lý.

F. Tài liệu tham khảo

- Next.js Documentation - Image Optimization
- Next.js Documentation - Font Optimization
- Next.js Documentation - Dynamic Imports
- Lighthouse Documentation

G. Bài tập nâng cao (tùy chọn)

1. **Tối ưu hóa ảnh động**: Sử dụng `next/image` với ảnh GIF hoặc video. Thử tích hợp thuộc tính `sizes` để tối ưu hóa trên các kích thước màn hình khác nhau.
2. **Tùy chỉnh font variable**: Sử dụng font có hỗ trợ variable (như `Inter`) và khai báo `variable` trong `next/font` để tạo hiệu ứng chuyển đổi mượt mà giữa các weight.
3. **Lazy-load thư viện lớn**: Tích hợp một thư viện phức tạp (ví dụ: `quill` cho trình soạn thảo văn bản) và lazy-load bằng `next/dynamic`. Kiểm tra hiệu năng trước và sau khi lazy-load.
4. **Phân tích hiệu năng**: Chạy Lighthouse trên dự án Blog trước và sau khi áp dụng các tối ưu hóa. Viết báo cáo ngắn (200 từ) so sánh các chỉ số FCP, LCP, CLS.