1. Detailed Description of the Solution

Overview: The project is a web application designed to deliver sequential training modules. Users must complete each module before advancing to the next. The application tracks the progress of users watching video content, ensuring they follow the intended sequence.

Technologies Used:

- **Frontend:**
    - **HTML**: Used to structure the content of the web pages.
    - **JavaScript**: Handles dynamic content and user interactions.
    - **Next.js**: A React framework that enables server-side rendering, API routes, and dynamic routing, ensuring fast and scalable web applications.
    - **Tailwind CSS**: Provides utility-first CSS for styling the application, allowing for responsive and consistent design.
    - **Video.js**: A popular open-source HTML5 video player library that enhances the video viewing experience by adding custom controls and features.
- **Backend:**
    - **Express.js**: A minimalist Node.js web application framework that provides robust features for building APIs and handling HTTP requests.
    - **PostgreSQL**: An advanced open-source relational database used to store videos and track user progress.

Key Components:

- **Video Management**: The application fetches video data from the database, ordering them sequentially. Videos are dynamically loaded using the useRouter from next/navigation for smooth navigation between modules.
- **Progress Tracking**: User progress is tracked and stored in a PostgreSQL database. The backend ensures that progress is updated when a video is watched, and only completed modules unlock the next one.
- **Sequential Access**: The system enforces that users must complete one module before accessing the next, which is handled by comparing progress values in the frontend.

Why These Choices?

- **Next.js** was chosen for its ability to handle both server-side rendering and static site generation, making it perfect for a scalable and SEO-friendly web application.
- **Express.js** provides a flexible and lightweight backend, allowing easy integration with PostgreSQL for handling user data and video progress.
- **Video.js** was selected to provide a robust and customizable video player experience, with support for various video formats and features.
- **Tailwind CSS** ensures that the application has a modern and responsive UI, with minimal effort and consistent design principles.

2. Code Snippets Demonstrating the Implementation

Fetching and Displaying Videos:

```
"use client";
import Link from 'next/link';
import { useEffect, useState } from 'react';

export default function Home() {
  const [videos, setVideos] = useState([]);
  const [progress, setProgress] = useState({}); // To store progress for each
video

  useEffect(() => {
    // Fetch videos data
    const fetchVideos = async () => {
      try {
        const response = await fetch('http://localhost:5000/api/videos');
        if (!response.ok) throw new Error('Failed to fetch videos');
        const data = await response.json();
        setVideos(data);

        // Fetch progress for each video
        const progressPromises = data.map(video =>
          fetch(`http://localhost:5000/api/videos/progress/video/${video.id}`)
            .then(res => res.json())
            .then(data => ({ id: video.id, progress: data.progress || 0 }))
            .catch(() => ({ id: video.id, progress: 0 }))
        );

        const progressData = await Promise.all(progressPromises);
        const progressMap = progressData.reduce((acc, { id, progress }) => {
          acc[id] = progress;
          return acc;
        }, {});
        console.log(progressMap)
```

```
        setProgress(progressMap);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchVideos();
  }, []);

  return (
    <div className="m-0 p-12 px-14 w-screen h-screen bg-cover bg-custom-
pattern">
      <h1 className="inline-flex text-3xl p-2 my-8 font-black mb-6 text-white
border">Training Modules</h1>
      <ul className="space-y-6">
        {videos.map(video => (
          <li key={video.id} className="bg-white shadow-lg text-white rounded-
lg overflow-hidden border border-gray-200 w-[30%] hover:shadow-xl transition-
shadow duration-300">
            <Link
              href={`/${video.id}`}
              className={`block p-4 ${
                video.id === 2 || progress[video.id - 2]?.progress === 1
                  ? progress[video.id]?.progress === 1
                    ? 'bg-green-500 hover:bg-green-700 pointer-events-auto'
                    : 'bg-white hover:bg-slate-300 pointer-events-auto'
                  : 'bg-slate-100 pointer-events-none opacity-50'
              }`}
            >
              <div className="flex justify-between items-center">
                <div className="flex flex-col">
                  <span className="text-lg font-semibold text-gray-
900">{video.title}</span>
                  <span className="text-sm text-gray-600 mt-1">
                    {progress[video.id] !== undefined
                      ? `Progress: ${(((progress[video.id].progress || 0) *
100).toFixed(2))}%`
                      : 'Loading...'}
                  </span>
                </div>
                <span className="text-gray-500">
                  <svg xmlns="http://www.w3.org/2000/svg" className="h-6 w-6"
fill="none" viewBox="0 0 24 24" stroke="currentColor">
                    <path strokeLinecap="round" strokeLinejoin="round"
strokeWidth={2} d="M5 12h14m-7 7l7-7-7-7" />
                  </svg>
                </span>
              </div>
```

```
            </Link>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Tracking and Updating Progress:

```
const saveProgress = async (userId, videoId, progress) => {
  const result = await query(
    'INSERT INTO progress (user_id, video_id, progress) VALUES ($1, $2, $3) ON
CONFLICT (user_id, video_id) DO UPDATE SET progress = $3',
    [userId, videoId, progress]
  );
  return result.rows[0];
};
```

```
const updateProgress = async (req, res) => {
  try {
    const { userId, videoId, progress } = req.body;
    await saveProgress(userId, videoId, progress);
    res.status(200).json({ message: 'Progress saved' });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};
```

3. Relevant Documentation or Diagrams

ER Diagram (Entity-Relationship Diagram):

- **Entities:**
    - **Videos**: video_id, title, sequence,
    - **Progress**: user_id, video_id, progress

Diagram Explanation:

- The Videos table stores details about each video, including the order in which they should be completed (sequence).

- The Progress table records each user's progress for each video, linking user_id with video_id.

PI Documentation:

- **GET /api/videos**
    - **Description**: Fetch all videos in sequence order.
    - **Response**: JSON array of video objects.
- **GET /api/videos/:id**
    - **Description**: Fetch details of a specific video by ID.
    - **Response**: JSON object of the video.
- **POST /api/progress**
    - **Description**: Update or save progress for a specific video.
    - **Request Body**: { userId, videoId, progress }
    - **Response**: JSON confirmation message.
- **GET /api/progress/video/:id**
    - **Description**: Fetch progress for a specific video by ID.
    - **Response**: JSON object containing the progress value.