# [프론트엔드] 기술 스택 정리 | React, Zustand, React Query, Tailwind CSS



## PWA에 적합한 이유로 이 기술 스택을 선택한 이유 요약

| 기술 | PWA 적합성 |
| --- | --- |
| **React** | PWA 요구 사항에 맞는 다양한 라이브러리와의 통합이 용이 |
| **Zustand** | 캐싱된 데이터를 관리하고 전역 상태를 효율적으로 처리 |
| **React Query** | 오프라인 상태 관리와 데이터 동기화에 최적화 |
| **Tailwind CSS** | 스타일 최적화와 반응형 디자인을 기본 제공 |

## 1. React

**왜 React인가?**

- **유연성**
  - React는 UI(화면)를 자유롭게 만들 수 있게 해준다.
    HTML과 JavaScript를 결합해서, 세부적인 부분까지 조정할 수 있다.
- **생태계**
  - PWA(Progressive Web App)와 관련된 여러 라이브러리(ex Workbox, React Router)가 잘 지원되기 때문에, PWA 개발에 유리하다.

## 2. Zustand

**왜 Zustand인가?**

- **간단한 사용법**
  - 직관적이고 사용하기 쉬운 API 제공, 다른 상태 관리 라이브러리(redux, ..)와 비교해봤을때 코드가 적고 간편하다.
- 경량성 및 성능
  - 매우 가벼운 라이브러리로, 번들 크기가 작아 초기 로딩 시간과 애플리케이션 성능에 긍정적인 영향을 미친다. 불필요한 렌더링을 최소화

**어떤 상황에 적합한가요?**

- 작은 프로젝트나 중간 규모의 프로젝트에서 매우 효과적
- 상태 관리가 간단하고 성능 최적화가 중요한 PWA에서도 유용하게 사용될 수 있다.

## 3. React Query

https://github.com/ssi02014/react-query-tutorial?tab=readme-ov-file#%EC%A3%BC%EC%9A%94-%EC%BB%A8%EC%85%89-%EB%B0%8F-%EA%B0%80%EC%9D%B4%EB%93%9C-%EB%AA%A9%EC%B0%A8

### 왜 React Query인가?

- **서버 상태 관리**
  - 서버에서 데이터를 받아오는 작업을 쉽게 관리할 수 있다.
  - 데이터가 서버에서 업데이트되거나 캐시되는 것을 자동으로 처리해준다.
- **PWA와의 통합**
  - PWA에서 중요한 **오프라인 상태 관리**와 **데이터 동기화**에 최적화되어 있어, 데이터가 끊겨도 이전에 캐시된 데이터를 사용할 수 있다.

### React Query vs Redux

- Redux는 서버 상태와 클라이언트 상태를 함께 관리하려면 코드가 많이 필요하지만, React Query는 서버 상태만 따로 관리하고, API 요청을 쉽게 처리할 수 있다.

▼ 예시 | 왼쪽 이미지 = react query 이용

```
// API 요청 함수
const getTodos = async () => {
  const response = await fetch('/api/todos');
  if (!response.ok) {
    throw new Error('데이터를 가져오는 데 실패했습니다.');
  }
  return response.json();
};

function Todos() {
  // useQuery로 GET 요청
  const { data, error, isLoading } = useQuery({
    queryKey: ['todos'], // 쿼리의 고유 키
    queryFn: getTodos,   // API 호출 함수
  });









  if (isLoading) return <p>로딩 중...</p>;
  if (error) return <p>오류: {error.message}</p>;

  return (
    <ul>
      {data.map((todo) => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
}
```

```
// API 요청 함수 (GET)
const getTodos = async () => {
  const response = await fetch('/api/todos');
  if (!response.ok) {
    throw new Error('데이터를 가져오는 데 실패했습니다.');
  }
  return response.json();
};

function Todos() {
  const [todos, setTodos] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchTodos = async () => {
      try {
        const data = await getTodos();
        setTodos(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    fetchTodos();
  }, []); // 컴포넌트 마운트 시 한 번만 실행


  if (loading) return <p>로딩 중...</p>;
  if (error) return <p>오류: {error}</p>;

  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
}
```

## 4. Tailwind CSS

### 왜 Tailwind CSS인가?

- **빠른 스타일링**
  - 스타일을 빠르고 효율적으로 적용할 수 있다.
  - className 속성으로 바로 디자인을 입힐 수 있어 작업 속도가 빠르다.
- **성능 최적화**
  - Tailwind는 `PurgeCSS`를 사용해서 실제로 사용되는 스타일만 포함시켜, 최종 코드의 크기를 줄여준다.

### Tailwind vs CSS-in-JS

- CSS-in-JS는 유연하지만, 성능 면에서 조금 더 부담이 있을 수 있다. Tailwind는 스타일을 정적으로 처리해 성능 면에서 더 유리하다.

▼ tailwind 코드 예시

```
import React from 'react';

const Button = () => {
  return (
    <button className="bg-blue-500 text-white py-2 px-4
      Click Me
    </button>
  );
};

export default Button;
```

- `bg-blue-500` : 중간 톤의 파란색 배경
- `text-white` : 텍스트 색상을 흰색으로 설정
- `py-2 px-4` : 수직 패딩 2, 수평 패딩 4
- `rounded` : 둥근 모서리
- `hover:bg-blue-700` : 호버 시 배경색이 어두운 파란색으로 변경
- `transition` : 스타일 변화에 애니메이션 적용
- `duration-300` : 애니메이션 지속 시간 0.3초

## 결론

- PWA의 주요 요구 사항(오프라인 지원, 데이터 캐싱, 반응형 디자인)에 부합
- 다른 선택지(redux, recoil, ..)도 있지만, 이 조합은 **개발 속도와 효율성을 극대화하면서도 확장성을 유지**할 수 있는 최적의 선택이라 생각