Consuming Web APIs with TypeScript

Advanced Fetch Options and Error Handling

Allen O'Neill

CTO / Senior Engineer, Microsoft Regional Director & MVP

@DataBytesAI | www.datalabs.io

Overview

In this section:

- Customizing Headers, Request Options, and Handling JSON Responses.
- Query Parameters, URL Manipulation, and Error Handling Strategies.
- Streaming Responses, Progress Tracking, and Creating Custom Types.
- Mapping HTTP Responses to Custom Types.
- Leveraging Type Declaration Files for Known Types

Customizing Headers and Request Options

Headers


HTTP Request
Method: GET
Path: /index.html
HTTP Response
Status: 200
Content-type: text/html

Key Reasons to Customize Headers

Authentication

Content-Type

Accept

Cache-Control

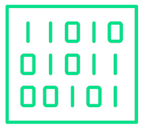
Other Metadata

Request Options

Key Reasons to Customize Request Headers

Precision in HTTP Method Selection

Fine-Tuning with Query Parameters



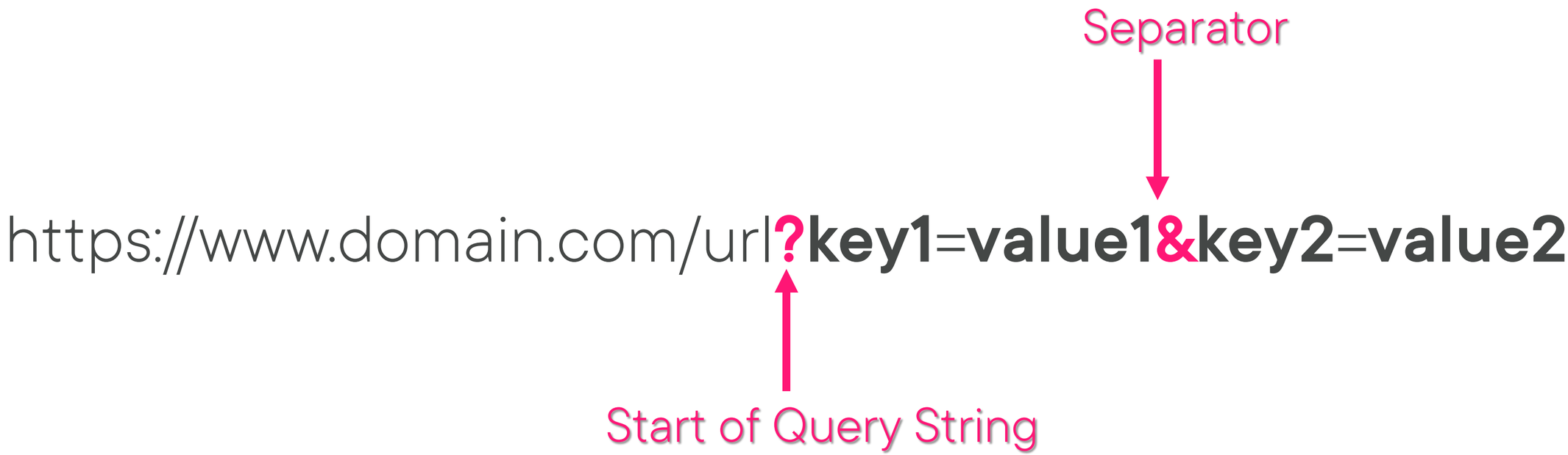
Crafting Data-Rich Request Bodies

Efficient Time Management with Timeouts

Controlling Response Formats

Query Parameters and URL Manipulation

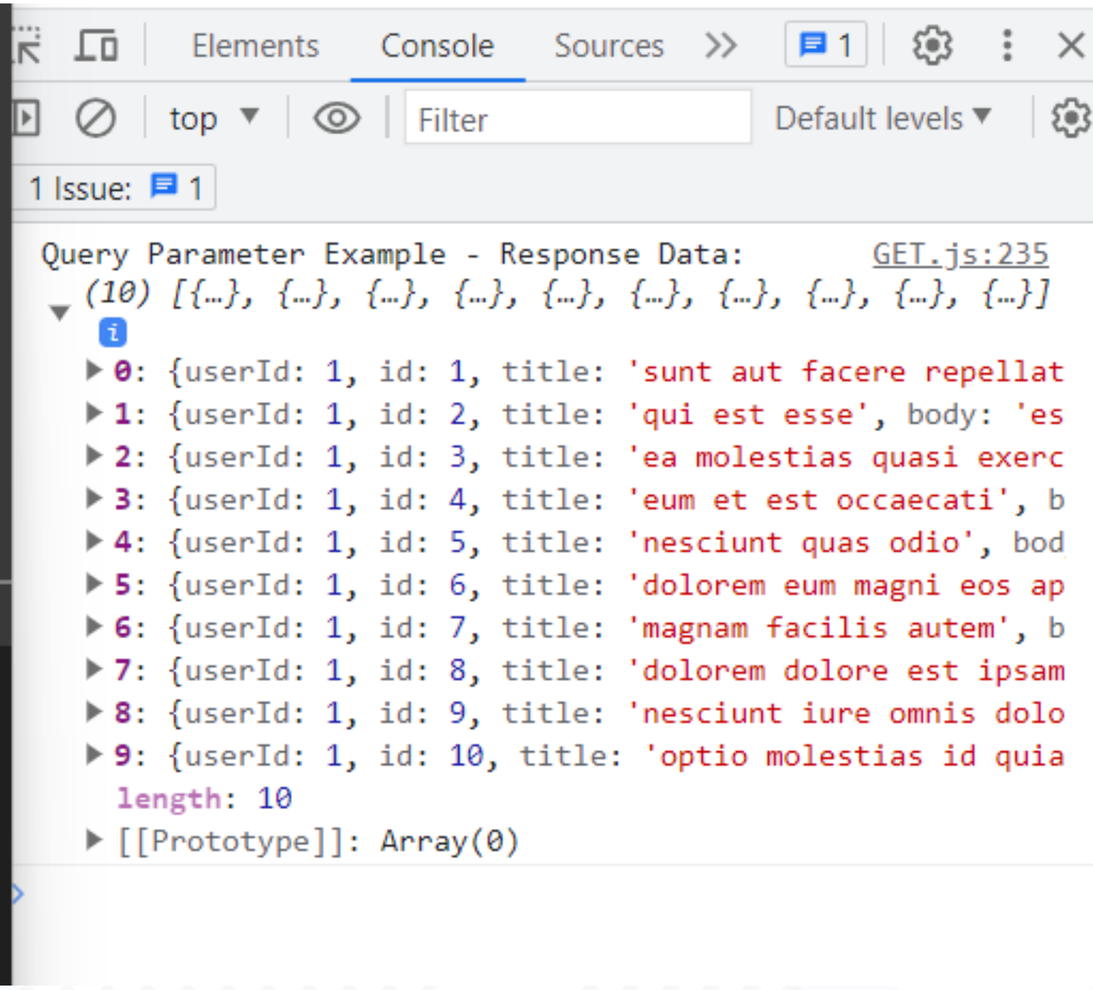
Query Parameters: Refining Data Requests



https://api.example.com/search?q=typescript&page=1

```
const userId = 1;

const apiUrl = 'https://jsonplaceholder.typicode.com/posts';

const queryUrl = `${apiUrl}?userId=${userId}`;

fetch(queryUrl)
  .then(response => response.json())

  .then(data => {
    console.log('Query Parameter Example - Response Data:', data);
  })

  .catch(error => {
    console.error('API Error:', error);
  });
```

```javascript
const userId = 1;

const apiUrl = 'https://jsonplaceholder.typicode.com/posts';

const queryUrl = `${apiUrl}?userId=${userId}`;

fetch(queryUrl)

  .then(response => response.json())

  .then(data => {
    console.log('Query Parameter Example - Response Data:', data);
  })

  .catch(error => {
    console.error('API Error:', error);
  });
```



POST Data Fields: Secure Data Submission

```
const data = {
username: 'johndoe',
password: 'mypassword'
};


const response = await fetch('https://api.example.com/users', {
method: 'POST',
headers: {
'Content-Type': 'application/json'
},
body: JSON.stringify(data)
});
```

```
const userData = {
    name: 'John Doe',
    email: 'john@example.com',
    password: 'securepassword123',
  };
```

```
  const apiUrl = 'https://jsonplaceholder.typicode.com/posts';
```

```
  fetch(apiUrl, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(userData),
  })
    .then(response => response.json())
    .then(data => {
      console.log('User Registration Response:', data);
    })
    .catch(error => {
      console.error('API Error:', error);
    });
```

Handling JSON Responses and Data Parsing

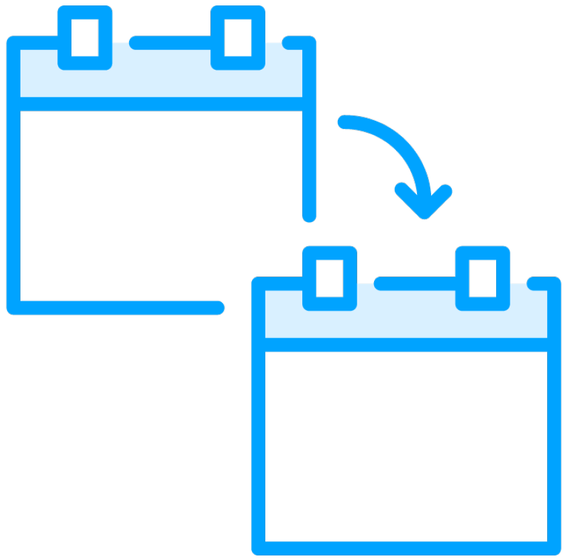
Understanding JSON

JavaScript Object Notation

JSON Object vs. JSON String

JSON Object	VS	JSON String
- Enclosed in curly braces {}		- Enclosed in double quotation marks “ ”
- Contains key-value pairs		- Appears as a plain text representation
- Represents structured data		- Used for data transmission and storage as text
- Behaves like a JavaScript object		- Behaves as a plain text string, not an object

Stringify



Converts a JavaScript object into a JSON string.

Example

```
const jsonObject = {
  name: "John Doe",
  age: 30,
  isStudent: false
};
```

```
const jsonString = JSON.stringify(jsonObject);
```

```
console.log(jsonString);
```

```
{
    "name": "John Doe",
    "age": 30,
    "isStudent": false
}
```

JSON Selectors/Query

. - Selects the current element.

[] - Selects an element from an array.

{} - Selects an element from an object.

```
{
  "location": "New York",
  "temperature": {
    "current": 75,
    "min": 68,
    "max": 82
  },
  "weather": "Sunny"
}
```

```
const jsonResponse = {
  "location": "New York",
  "temperature": {
    "current": 75,
    "min": 68,
    "max": 82
  },
  "weather": "Sunny"
};

const currentTemperature = jsonResponse.temperature.current;

const weatherCondition = jsonResponse.weather;

console.log("Current Temperature:", currentTemperature);
console.log("Weather Condition:", weatherCondition);
```

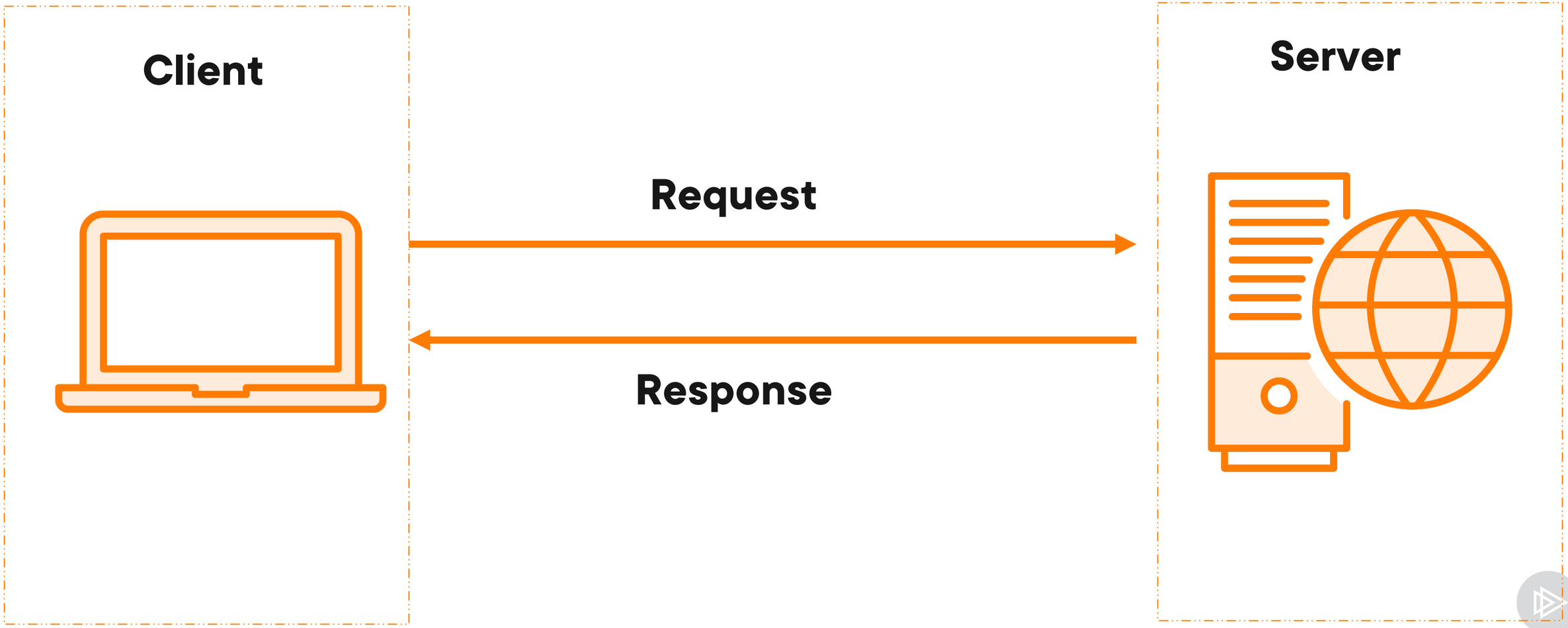
JSON Selectors/Query

```
const jsonResponse = {
  "location": "New York",
  "temperature": {
    "current": 75,
    "min": 68,
    "max": 82
  },
  "weather": "Sunny"
};

const currentTemperature = jsonResponse.temperature.current;
const weatherCondition = jsonResponse.weather;

console.log("Current Temperature:", currentTemperature);
console.log("Weather Condition:", weatherCondition);
```

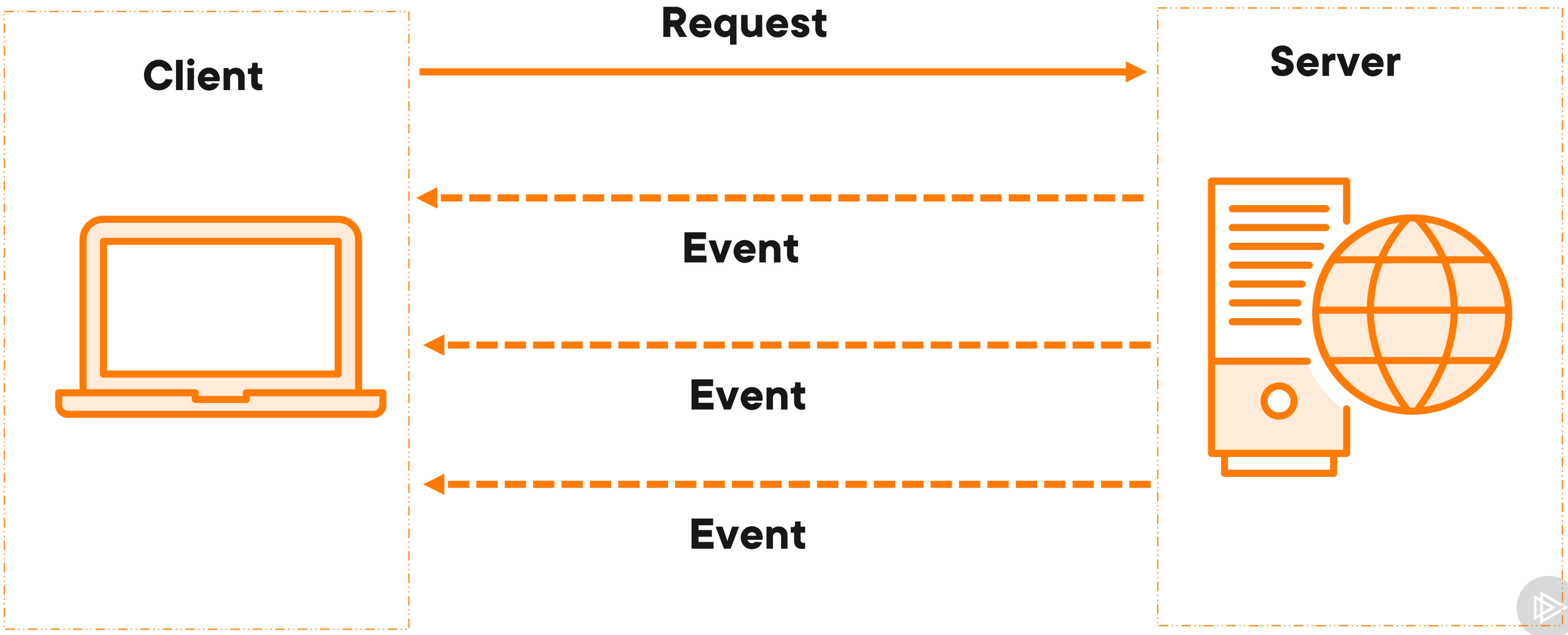
```
Current Temperature: 75
Weather Condition: Sunny
```

Streaming Responses and Progress Tracking

Streaming vs. Server-Client Conversations



Streaming vs. Server-Client Conversations



Progress Tracking

```
const imageUrl = 'https://fetch-progress.anthum.com/30kbps/images/sunrise-baseline.jpg';

async function trackProgress(url: string) {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error('Download failed with status ' + response.status);
    }

    const contentLength = response.headers.get('Content-Length');
    if (!contentLength) {
      throw new Error('Content-Length header not found in response.');
    }

    const totalSize = parseInt(contentLength, 10);
    let receivedSize = 0;

    const downloadInterval = setInterval(() => {
      console.log('Received Progress:', ((receivedSize / totalSize) * 100).toFixed(2) + '%');

      if (receivedSize === totalSize) {
        clearInterval(downloadInterval);
        console.log('Download complete.');
      }
    }, 1000);

    const reader = response.body.getReader();

    while (true) {
      const { done, value } = await reader.read();

      if (done) {
        break;
      }

      receivedSize += value.length;
    }
  } catch (error) {
    console.error('Error:', error.message);
  }
}

trackProgress(imageUrl);
```

```
const imageUrl = 'https://fetch-progress.anthum.com/30kbps/images/sunrise-baseline.jpg';

async function trackProgress(url: string) {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error('Download failed with status ' + response.status);
    }

    const contentLength = response.headers.get('Content-Length');
    if (!contentLength) {
      throw a Error('Content-Length header not found in response.');
    }

    const totalSize = parseInt(contentLength, 10);
    let receivedSize = 0;

    const downloadInterval = setInterval(() => {
      console.log('Received Progress:', ((receivedSize / totalSize) * 100).toFixed(2) + '%');

      if (receivedSize === totalSize) {
        clearInterval(downloadInterval);
        console.log('Download complete.');
      }
    }, 1000);
```

```
    const reader = response.body.getReader();
```

```
    while (true) {
      const { done, value } = await reader.read();

      if (done) {
        break;
      }
```

```
      receivedSize += value.length;
```

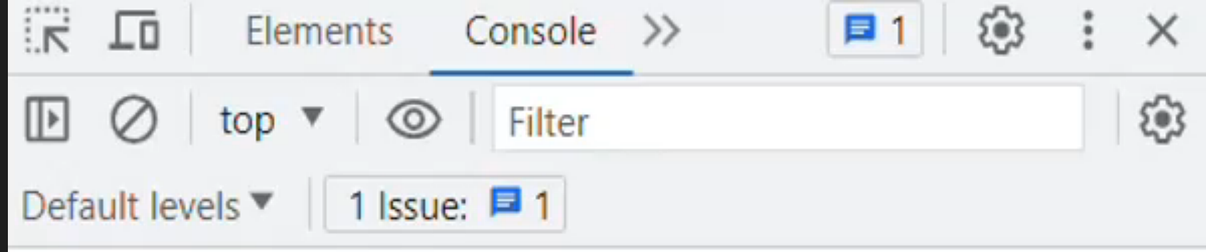
```
    }
  }
```

```
  catch (error) {
    console.error('Error:', error.message);
```

```
  }
}
```

```
trackProgress(imageUrl);
```

```ts
const imageUrl = 'https://fetch-progress.anthum.com/30kbps/images/sunrise-baseline.jpg';

async function trackProgress(url: string) {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error('Download failed with status ' + response.status);
    }

    const contentLength = response.headers.get('Content-Length');
    if (!contentLength) {
      throw new Error('Content-Length header not found in response.');
    }

    const totalSize = parseInt(contentLength, 10);
    let receivedSize = 0;

    const downloadInterval = setInterval(() => {
      console.log('Received Progress:', ((receivedSize / totalSize) * 100).toFixed(2) + '%');

      if (receivedSize === totalSize) {
        clearInterval(downloadInterval);
        console.log('Download complete.');
      }
    }, 1000);

    const reader = response.body.getReader();

    while (true) {
      const { done, value } = await reader.read();

      if (done) {
        break;
      }

      receivedSize += value.length;
    }
  } catch (error) {
    console.error('Error:', error.message);
  }
}

trackProgress(imageUrl);
```

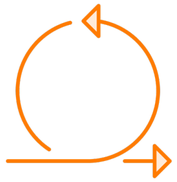


Error Handling Strategies and Status Codes

Common Error Handling Strategies

Try-Catch

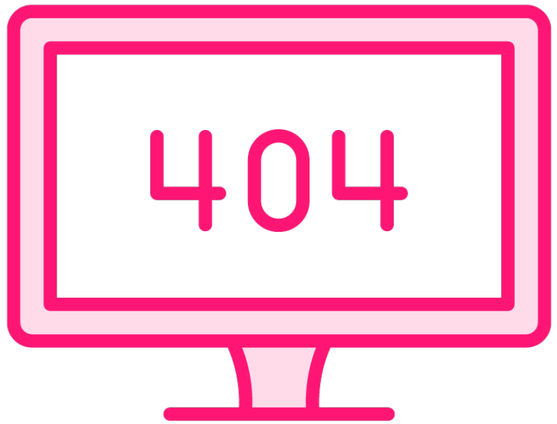
Status Codes



Retry Mechanisms

Fallback Mechanisms

HTTP Response Status Codes



1xx Informational Status Codes

100 (Continue)

Informs the client to continue sending the request body.

101 (Switching Protocols)

Indicates a change in protocol; often seen in WebSocket connections.

2xx Success Status Codes

200 (OK)

The request was successful, and the server responds with the requested data.

201 (Created)

Typically used after a successful POST request to indicate that a new resource has been created.

204 (No Content)

Signals successful execution with no response body, often seen in DELETE requests.

3xx Informational Status Codes

301 (Moved Permanently)

Informs the client that the resource has moved permanently to a new location.

304 (Not Modified)

Indicates that the client's cached version is still valid; no need to re-fetch.

4xx Success Status Codes

400 (Bad Request)

Typically used for malformed requests.

403 (Forbidden)

Denies access to a resource, often due to insufficient permissions.

404 (Not Found)

The requested resource could not be found on the server.

5xx Success Status Codes

500 (Internal Server Error)

502 (Bad Gateway)

504 (Gateway Timeout)

```
fetch('https://jsonplaceholder.typicode.com/posts/1000')
  .then(response => {
    if (!response.ok) {
      throw new Error('Resource not found');
    }
    return response.json();
  })
  .then(data => {
    console.log('Data:', data);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

Storing Data Retrieved from Public APIs

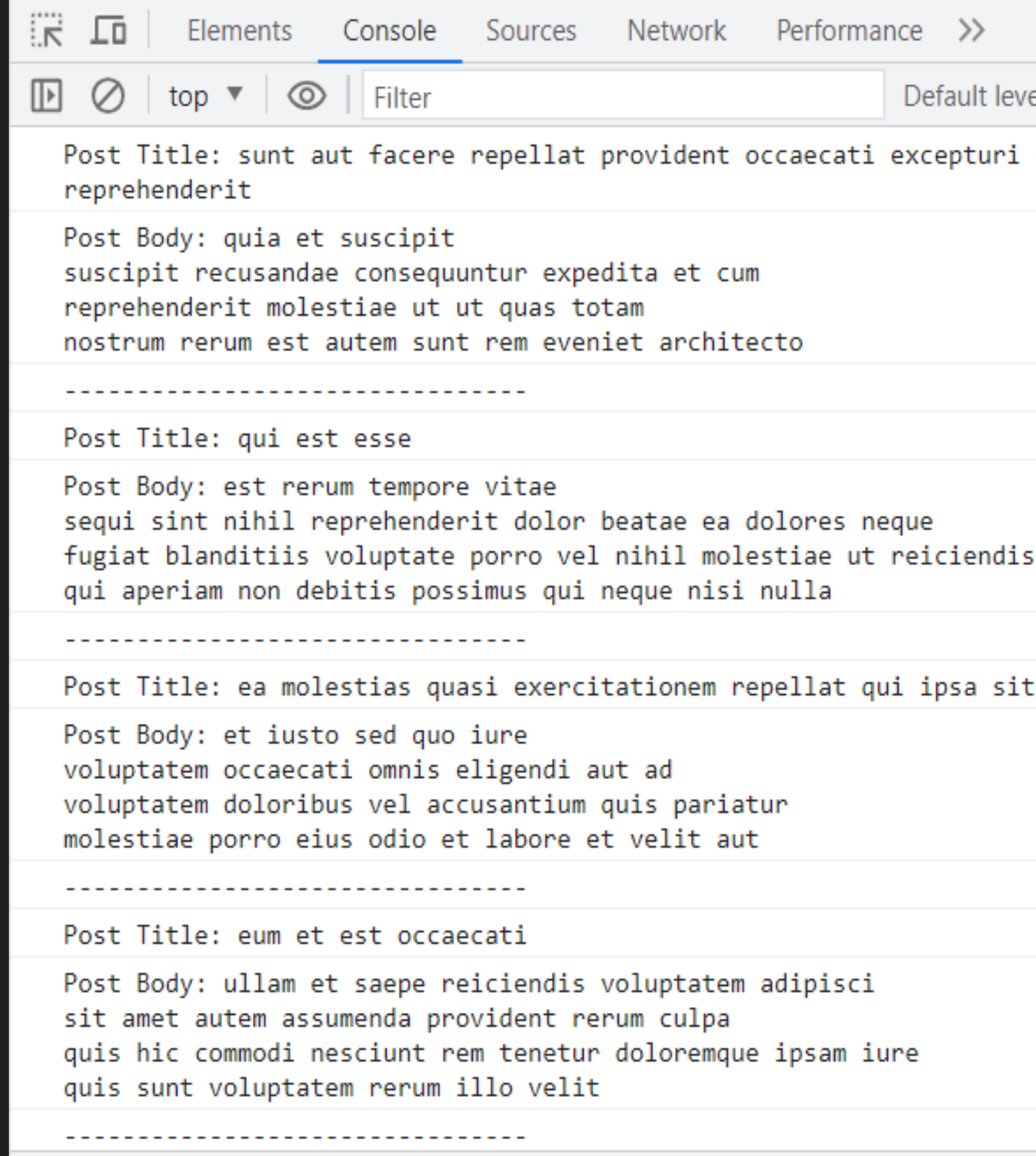
```typescript
interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}
```

```typescript
async function fetchPosts(): Promise<Post[]> {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data: Post[] = await response.json();
    return data;
  } catch (error) {
    console.error('API Error:', error);
    throw error;
  }
}
```

```typescript
async function displayPosts() {
  try {
    const posts: Post[] = await fetchPosts();
    posts.forEach(post => {
      console.log(`Post Title: ${post.title}`);
      console.log(`Post Body: ${post.body}`);
      console.log('------------------------------');
    });
  } catch (error) {
    console.error('Error fetching and displaying posts:', error);
  }
}
```

```typescript
displayPosts();
```

```typescript
interface Post {
  userId: number;
  id: number;
  title: string;
  body: string;
}

async function fetchPosts(): Promise<Post[]> {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data: Post[] = await response.json();
    return data;
  } catch (error) {
    console.error('API Error:', error);
    throw error;
  }
}

async function displayPosts() {
  try {
    const posts: Post[] = await fetchPosts();
    posts.forEach(post => {
      console.log(`Post Title: ${post.title}`);
      console.log(`Post Body: ${post.body}`);
      console.log('---------------------------------');
    });
  } catch (error) {
    console.error('Error fetching and displaying posts:', error);
  }
}

displayPosts();
```



Mapping HTTP Responses to Custom Types

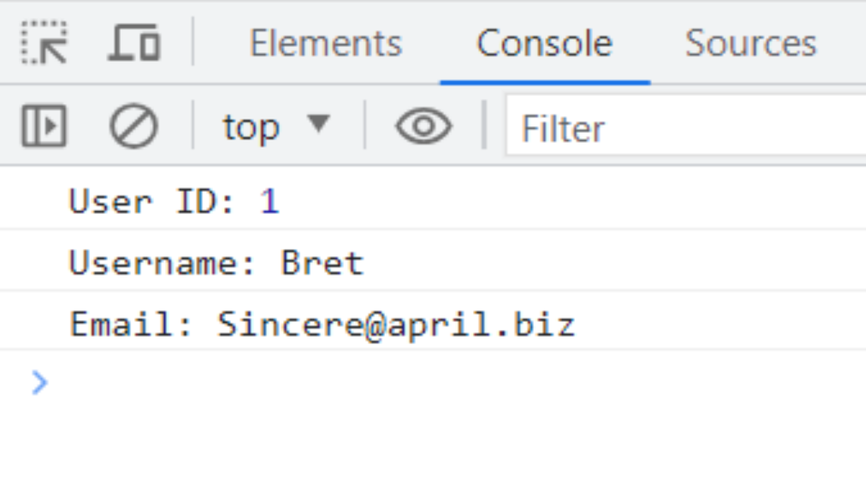
- Enhances code readability with concise property names in responses.

- Averts format errors by ensuring data integrity.

- Boosts performance by reducing JSON parsing redundancy.

```typescript
interface User {
    id: number;
    username: string;
    email: string;
}

async function fetchUserData(): Promise<User> {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/users/1');
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const userData: User = await response.json();
        return userData;
    } catch (error) {
        console.error('API Error:', error);
        throw error;
    }
}

async function getUserInfo() {
    try {
        const user: User = await fetchUserData();
        console.log('User ID:', user.id);
        console.log('Username:', user.username);
        console.log('Email:', user.email);
    } catch (error) {
        console.error('Error fetching user data:', error);
    }
}

getUserInfo();
```

```typescript
interface User {
    id: number;
    username: string;
    email: string;
}

async function fetchUserData(): Promise<User> {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/users/1');
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        const userData: User = await response.json();
        return userData;
    } catch (error) {
        console.error('API Error:', error);
        throw error;
    }
}

async function getUserInfo() {
    try {
        const user: User = await fetchUserData();
        console.log('User ID:', user.id);
        console.log('Username:', user.username);
        console.log('Email:', user.email);
    } catch (error) {
        console.error('Error fetching user data:', error);
    }
}

getUserInfo();
```

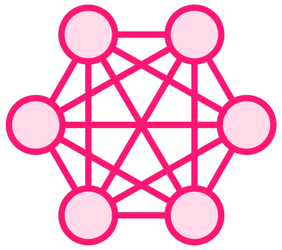


Leveraging Type Declaration Files

Importance of Mapping Response Objects to Known Types

Enhanced Code Readability: Clear and Concise Property Names

Error Prevention: Ensuring Correct Data Format



Performance Boost: Avoiding Redundant JSON Parsing

Practical Implementation

Key Concepts

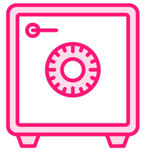
- Sending Custom Headers in API Requests
- Mapping HTTP Responses to Custom Types

```
const fetchDataWithCustomHeader = async () => {
  try {
    const customHeaders = new Headers();
    customHeaders.append('Authorization', 'Bearer yourAccessToken');
    const response = await fetch(`http://localhost:4000/opensky-
local?fileId=data${counterId}`, {
      method: 'GET',
      headers: customHeaders,
    });
    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}`);
    }

    const jsonData = await response.json();
    setFlightData(jsonData);
  } catch (error) {
    console.error('Error:', error);
  }
};
```

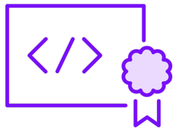
```
// Define a custom type
interface CustomApiResponse {
  data: string;
  customField: number;
}
```

```
// Fetch and handle the response
const fetchDataWithCustomMapping = async () => {
  try {
    const response = await fetch(`http://localhost:4000/opensky-
local?fileId=data${counterId}`, {
      method: 'GET',
    });

    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}`);
    }

    const jsonData: CustomApiResponse = await response.json();

    setFlightData(jsonData);
  } catch (error) {
    console.error('Error:', error);
  }
};
```

Importance of Security and Authorization



Protecting Confidential Information



Compliance and Trust

Mitigating Risks

User Trust

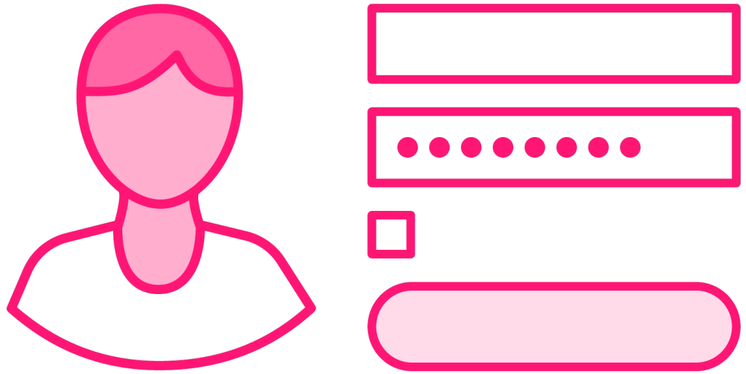
Overview

In this section:

- Basics of API Authentication and Authorization
- Implementing Token-Based Authentication
- Securing Sensitive Data in Transit and at Rest
- Cross-Origin Resource Sharing (CORS) Considerations
- Handling User Authentication and Authorization

Basics of API Authentication and Authorization

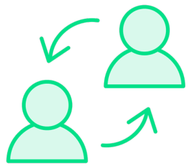
Authentication: The Proof of Identity



Importance of Authentication

Secure Access

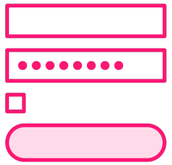
User Identity



Accountability

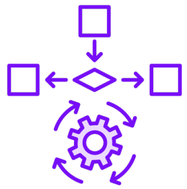
Customization

Types of Authentication



Basic Authentication

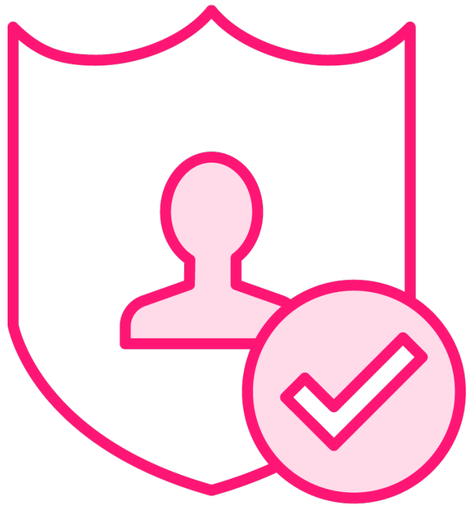
Token-Based Authentication



OAuth

API Keys

Authorization



Importance of Authorization

Access Control

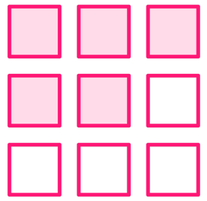
Data Protection

Compliance

Security

Authorization Techniques

Role-Based Access Control (RBAC)



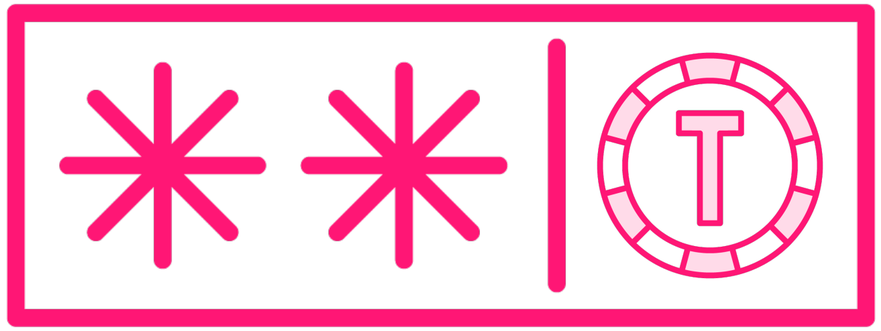
Attribute-Based Access Control (ABAC)

OAuth Scopes

Custom Logic

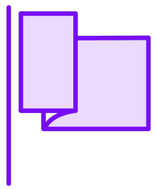
Implementing Token Based Authentication

Understanding Token-Based Authentication



Advantages of Token Based Authentication

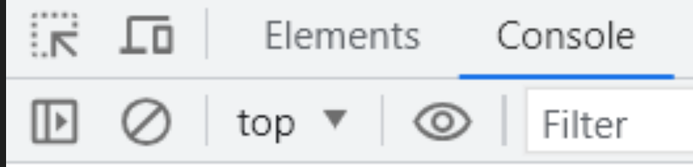
Enhanced Security



Statelessness

Efficiency

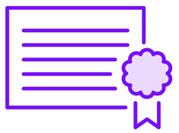
```
const serverUrl = 'https://api.github.com';

async function performTokenAuthentication(token: string): Promise<string | null> {
  try {
    const response = await fetch(`${serverUrl}/user`, {
      method: 'GET',
      headers: {
        'Authorization': `token ${token}`,
        'Content-Type': 'application/json',
      },
    });

    if (!response.ok) {
      throw new Error('Authentication failed.');
    }

    const data = await response.json();
    const username = data.login;

    return username;
  } catch (error) {
    console.error('Authentication Error:', error);
    return null;
  }
}
const githubToken = 'Use_Your_GitHub_Token_Here';

performTokenAuthentication(githubToken)
  .then((username) => {
    if (username) {
      console.log('Authentication successful. \nUsername:', username);
    } else {
      console.log('Authentication failed.');
    }
  })
  .catch((error) => {
    console.error('Authentication Error:', error);
  });
```



Securing Sensitive Data in Transit and at Rest

Importance of Data Security

Data Privacy

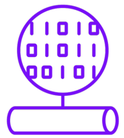


Regulatory Compliance

User Trust

Business Reputation

Ways to Secure Sensitive Data



Encryption

Authentication and Authorization

Access Control

Secure Storage

Secure Key Management

Ways to Secure Sensitive Data

Data Masking

Regular Auditing and Monitoring

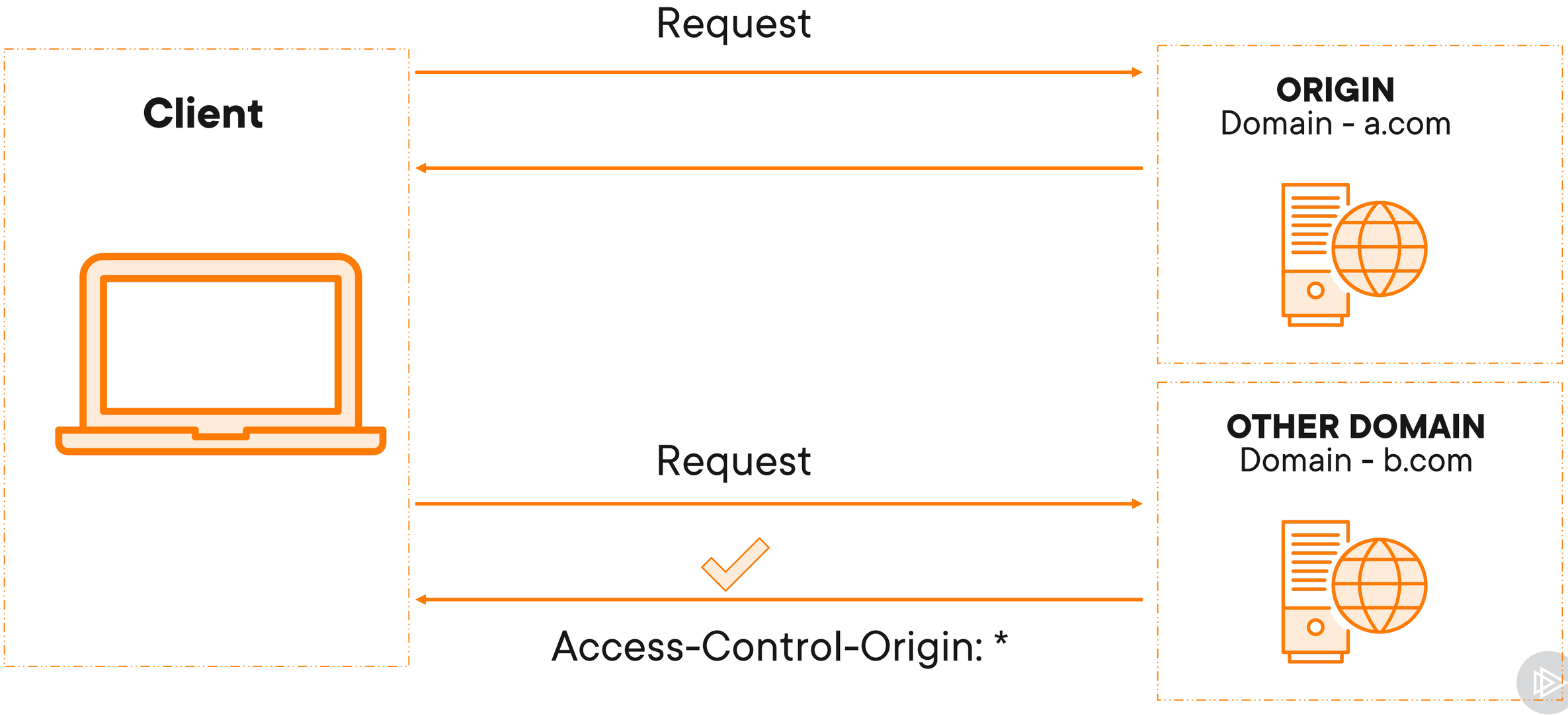
Secure File Uploads

API Security

Security Training

Cross-Origin Resource Sharing (CORS) Considerations

Understanding CORS

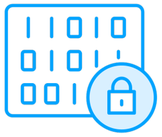


Ways to Secure Sensitive Data

Same-Origin Policy Enforcement

Security

Flexibility and Power



Data Protection

Controlled Access

Considerations for Implementing CORS

Origin Whitelisting

Configuration of HTTP Errors

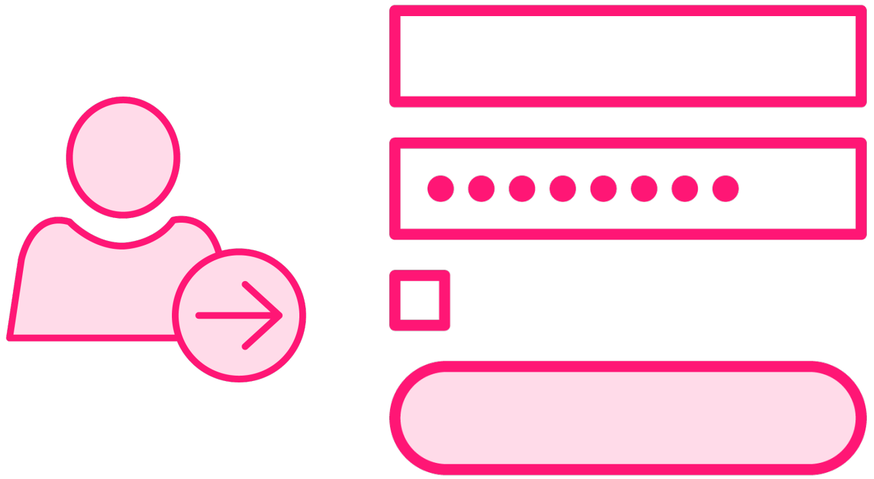
Handling Preflight Requests

Authentication and Cookies

Error Handling

Practical Implementation

Secure Access



```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

```
const users = [
  { username: 'admin', password: 'admin' },
];
```

```
let isAuthenticated = false;
```

```
// Authentication middleware
const authenticate = (req, res, next) => {
```

```
  if (isAuthenticated) {
    // User is authenticated, proceed to the next middleware or route
handler.
    next();
  }
```

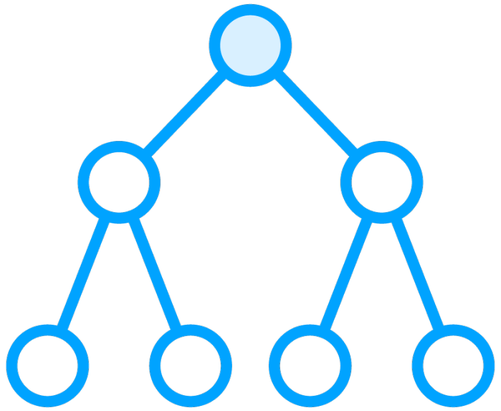
```
  else {
    res.status(401).json({ error: 'Unauthorized' });
  }
};
```

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  const user = users.find((u) => u.username === username && u.password === password);

  if (user) {
    isAuthenticated = true;
    res.status(200).json({ message: 'Login successful' });
  }

  else {
    isAuthenticated = false;
    res.status(401).json({ message: 'Login failed' });
  }
};
```

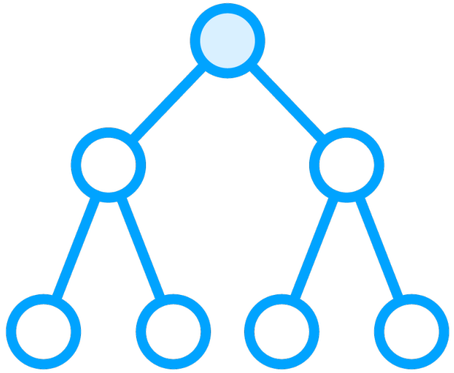
```tsx
import React, { useState } from 'react';
import './login.css';

type LoginProps = {
  onLogin: (username: string, password: string) => void;
  error: string;
};

const Login: React.FC<LoginProps> = ({ onLogin, error }) => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = () => {
    onLogin(username, password);
  };

  const saveLoginCredentials = () => {
    // Save the username and password to local storage
    localStorage.setItem('username', username);
    localStorage.setItem('password', password);
  };

  const handleCredentialInput = () => {
    // Get saved credentials from local storage
    const savedUsername = localStorage.getItem('username');
    const savedPassword = localStorage.getItem('password');

    // If saved credentials exist, fill in the input fields
    if (savedUsername && savedPassword) {
      setUsername(savedUsername);
      setPassword(savedPassword);
    }
  };
```

```
 React.useEffect(() => {
    handleCredentialInput();
  }, []);

  return (
    <div className='login-container'>
      <div>
        <div className='login-wrapper'>
          <h2>Login</h2>
          <div className='login-item'>
            <input
              type="text"
              placeholder="Username"
              value={username}
              onChange={(e) => setUsername(e.target.value)}
            />
          </div>
          <div className='login-item'>
            <input
              type="password"
              placeholder="Password"
              value={password}
              onChange={(e) => setPassword(e.target.value)}
            />
          </div>
          <div className='login-item'>
            <button onClick={() => { handleLogin(); saveLoginCredentials(); }}>Login</button>
          </div>
        </div>
      </div>
      {error && <p>{error}</p>}
    </div>
  );
};

export default Login;
```

Nested Data Structures

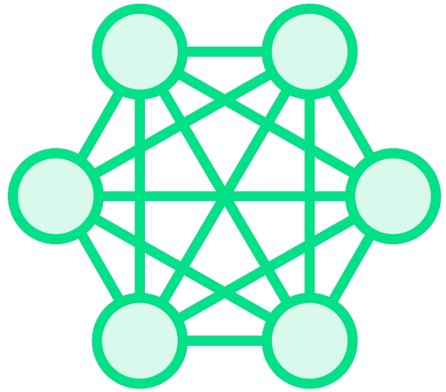


Visualizing Nested Data Structures



Tree

A tree is a hierarchical data structure with nodes and edges connected in a parent-child relationship.



Graph

A graph is a data structure where nodes are connected by edges. Graphs lack strict hierarchy, fostering complex relationships.

Examples of Nested Data Structures

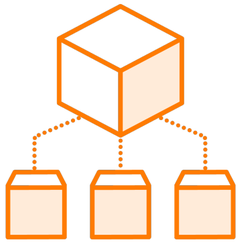
JSON

```
{
  "person": {
    "name": "John",
    "address": {
      "street": "123 Main St",
      "city": "Anytown"
    },
    "hobbies": ["reading", "gaming"]
  }
}
```

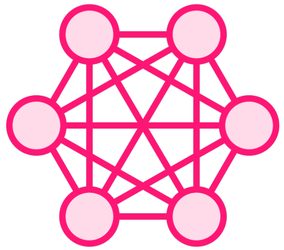
XML

```
<person>
  <name>John</name>
  <address>
    <street>123 Main St</street>
    <city>Anytown</city>
  </address>
  <hobbies>
    <hobby>reading</hobby>
    <hobby>gaming</hobby>
  </hobbies>
</person>
```

Importance of Nested Data Structures



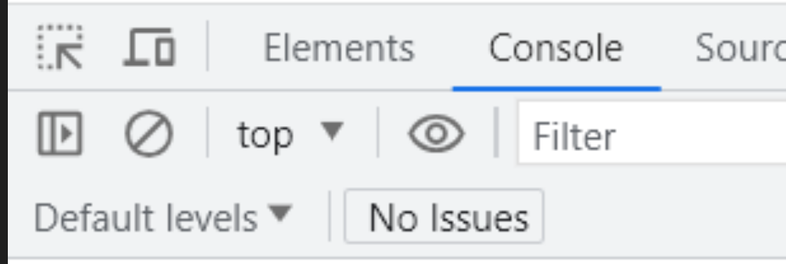
Organize data in a more hierarchical way.

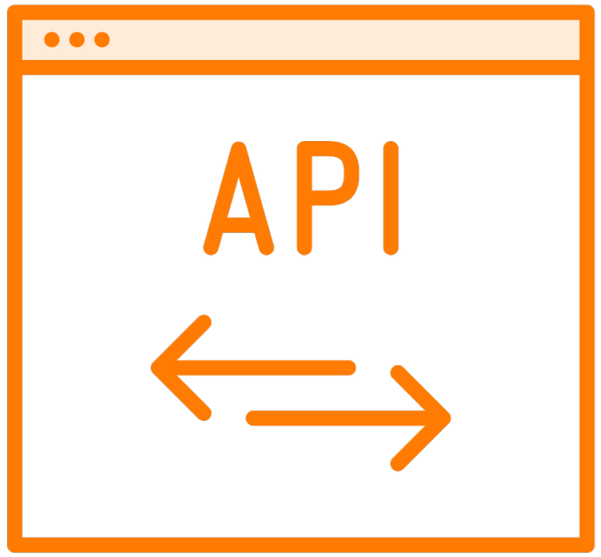


To represent complex relationships.

To improve performance of algorithms.

```
const jsonResponse = {
    "books": [
      {
        "title": "The Catcher in the Rye",
        "author": {
          "name": "J.D. Salinger",
          "birthYear": 1919
        }
      },
      {
        "title": "To Kill a Mockingbird",
        "author": {
          "name": "Harper Lee",
          "birthYear": 1926
        }
      }
    ]
  }

const first_name = jsonResponse.books[0].author.name;
const first_year = jsonResponse.books[0].author.birthYear;

console.log(`${first_name} was born in ${first_year}`)
```

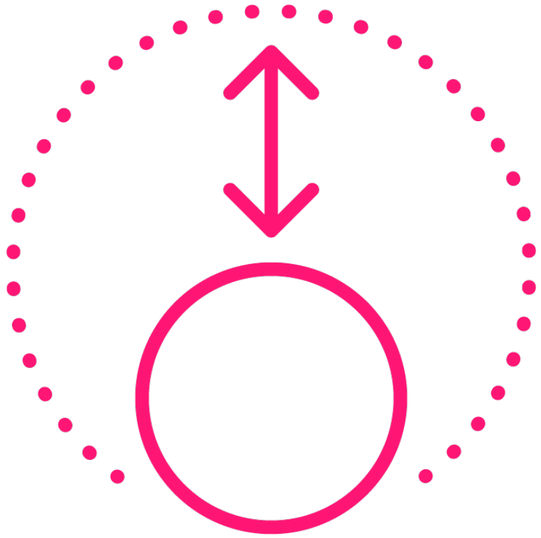




- Recursive Algorithm
- Libraries

Normalizing and Transforming API Responses

Normalizing and Transforming

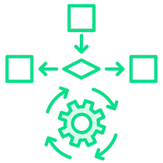


Importance of Normalizing and Transforming API Responses

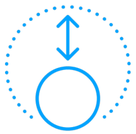
Simplifying Complex Data

Improving Performance

Enhancing Code readability



Reducing Redundancy



Aligning with Application Needs

Normalizing and Transforming API Responses

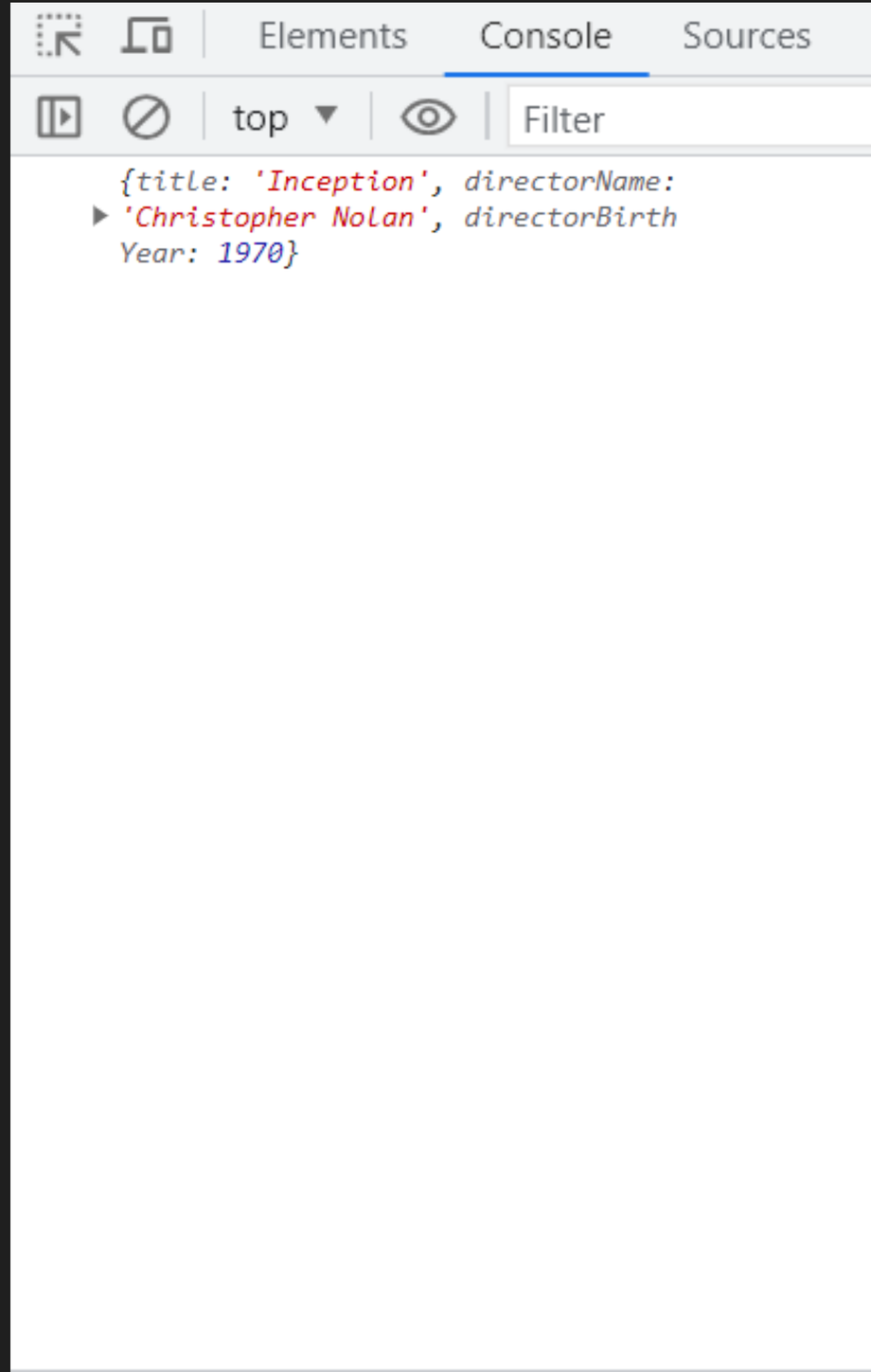
Nested Data Structure

```
{
  "title": "Inception",
  "director": {
    "name": "Christopher Nolan",
    "birthYear": 1970
  }
}
```

Flattened Data Structure

```
{
    "title": "Inception",
    "directorName": "Christopher Nolan",
    "directorBirthYear": 1970
  }
```

```
const apiResponse = {
    title: "Inception",
    director: {
      name: "Christopher Nolan",
      birthYear: 1970,
    },
  };


const transformedData = {
    title: apiResponse.title,
    directorName: apiResponse.director.name,
    directorBirthYear: apiResponse.director.birthYear,
  };


console.log(transformedData);
```

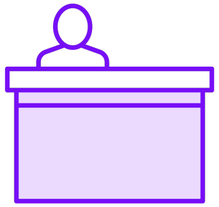


Pagination and Handling Paginated API Data

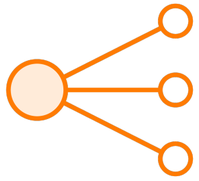
Pagination

Importance of Pagination

Improved Performance



User Experience



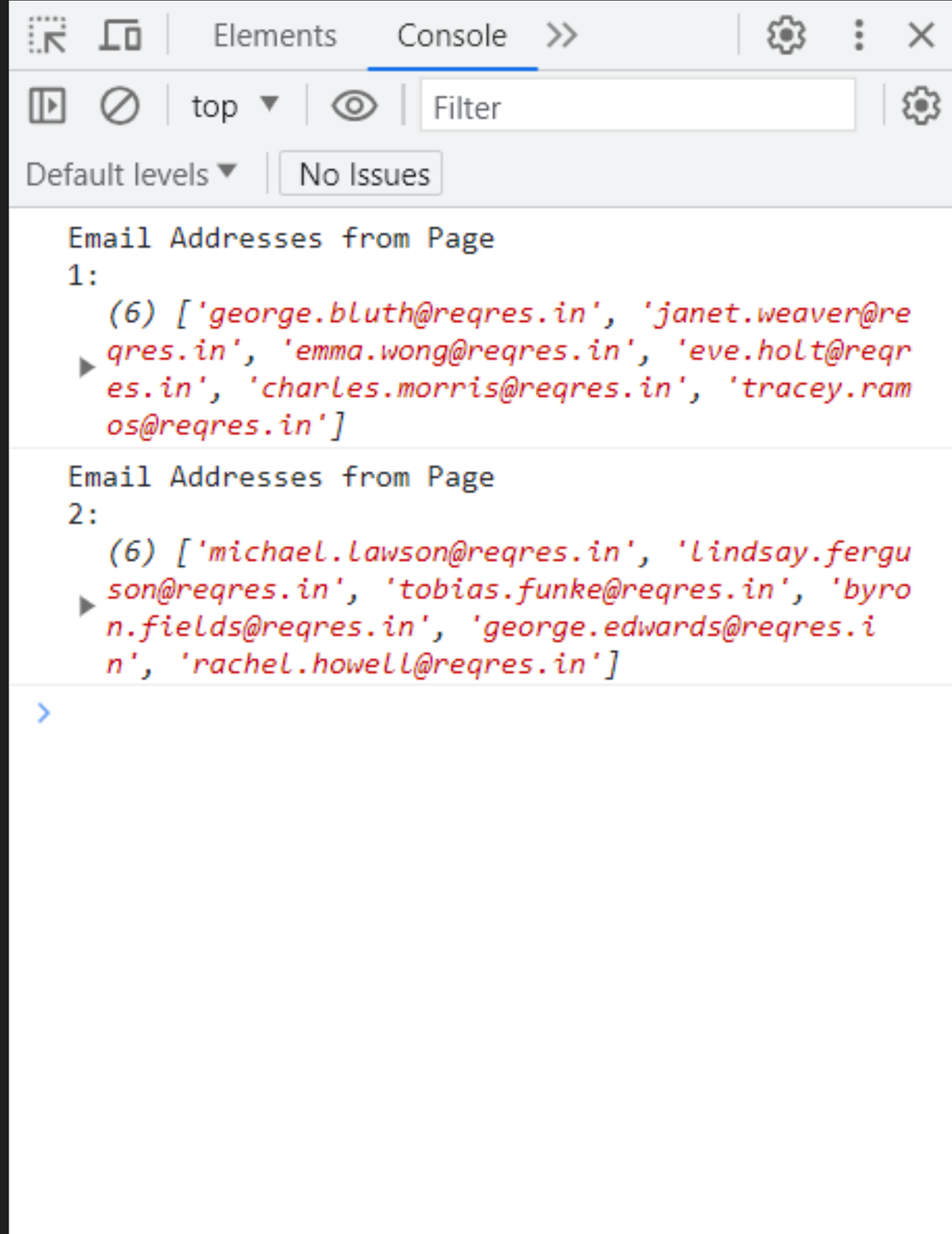
Reduced Server Load

```typescript
async function fetchUserData(page: number): Promise<string[]> {
    const url = `https://reqres.in/api/users?page=${page}`;
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error(`Error fetching data from ${url}`);
    }

    const data = await response.json();
    const emails = data.data.map((user: any) => user.email);

    return emails;
  }
```

```typescript
  async function fetchAllUserEmails(pages: number): Promise<string[][]> {
    const userEmailsByPage: string[][] = [];

    for (let page = 1; page <= pages; page++) {
      const emails = await fetchUserData(page);
      userEmailsByPage.push(emails);
    }

    return userEmailsByPage;
  }
```

```typescript
  const totalPages = 2;
```

```typescript
  fetchAllUserEmails(totalPages)
    .then((userEmailsArray) => {
      userEmailsArray.forEach((emails, page) => {
        console.log(`Email Addresses from Page ${page + 1}:`, emails);
      });
    })
    .catch((error) => {
      console.error('Error:', error);
    });
```



Strategies for Dealing with Large Datasets

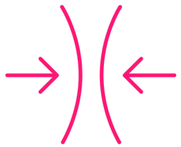
Data Pagination

Lazy Loading

Data Filtering

Data Caching

Strategies for Dealing with Large Datasets



Compression

Optimized Requests

Server-Side Processing

Progressive Loading

Browser Caching Mechanisms and Cache-Control Headers

Browser Caching

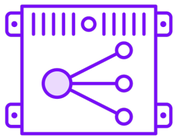
Importance of Browser Caching

Improves Performance

Reduces Bandwidth Usage

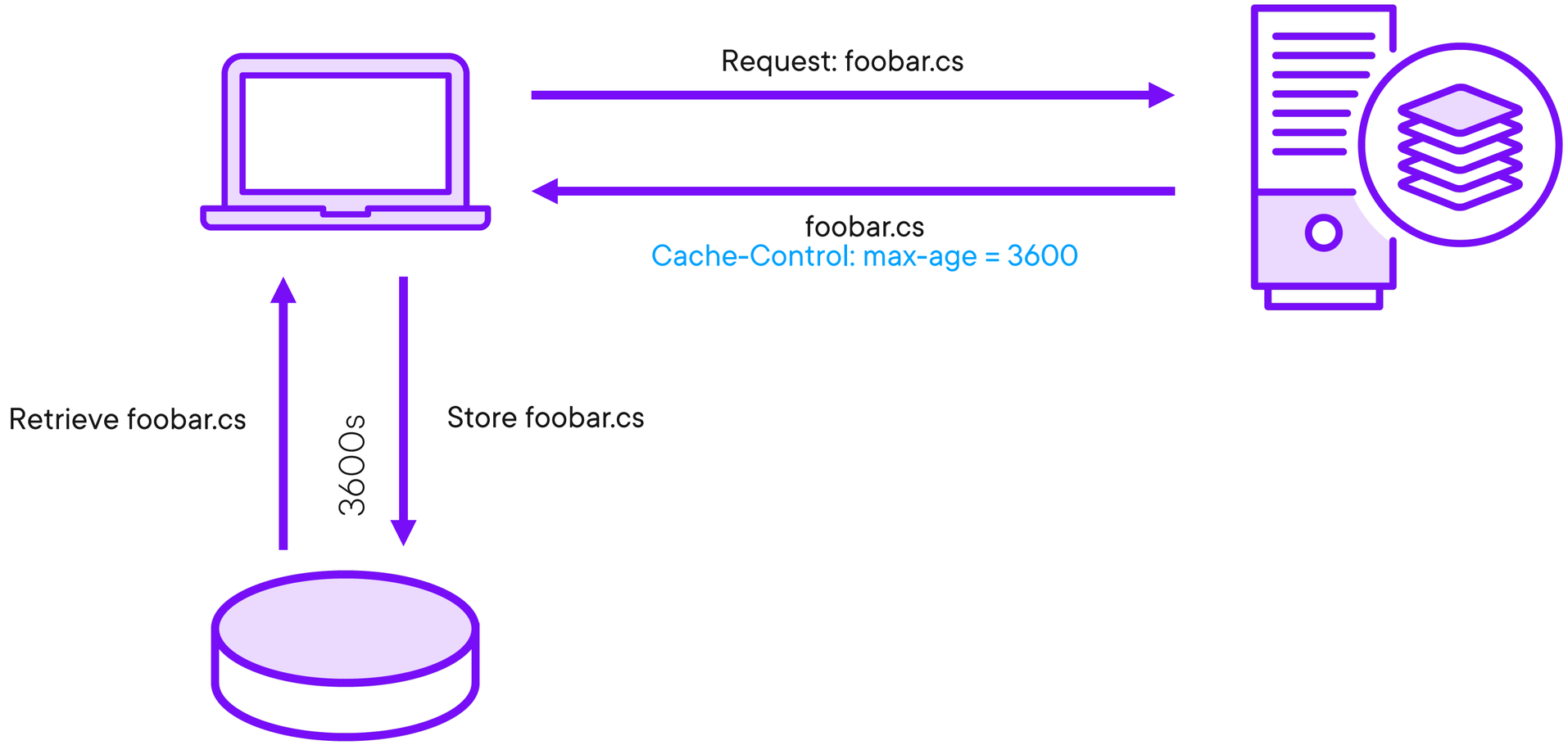
Improves Reliability

Enhances User Experience



Reduces Load on Servers

Cache Control Headers



Example

```
const response = await fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Cache-Control': 'max-age=3600’,
  },
});

const data = await response.json();
```

Client-Side Caching for API responses

```
async function fetchDataFromSWAPI(): Promise<any> {
    const cacheKey = 'swapiData';

    const cachedData = localStorage.getItem(cacheKey);

    if (cachedData) {

      return JSON.parse(cachedData);
    }

    else {
      const response = await fetch('https://swapi.dev/api/people/1/');
      const data = await response.json();

      localStorage.setItem(cacheKey, JSON.stringify(data));

      return data;
    }
  }

  async function main() {
    try {
      const swapiData = await fetchDataFromSWAPI();
      console.log('SWAPI Data:', swapiData);
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  }

  main();
```

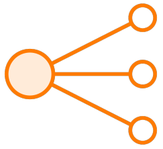
Service Workers and Offline Capabilities

Understanding Service Workers

Importance of Service Workers

Offline Capability

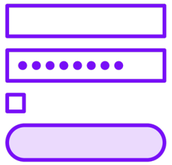
Improved Performance



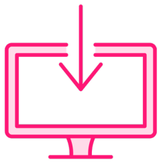
Reduced Server Load

Background Synchronization

Implementing Service Workers



Registration

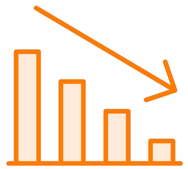


Installation

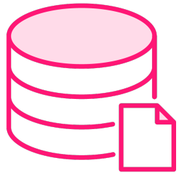
Activation

Fetching and Caching

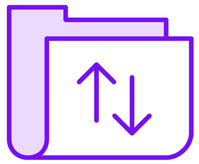
Performance Optimization Strategies



Reduce the Number of Requests



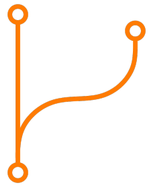
Implement Caching



Optimize Data Transfer

Performance Optimization Strategies

Implement Rate Limiting



Parallelize Requests

Monitor and Analyze Performance

Practical Implementation

Client-Side Caching

```
// to implement client-side caching for login we should update login.tsx
component
```

```
const saveLoginCredentials = () => {
  localStorage.setItem('username', username);
  localStorage.setItem('password', password);
};
```

```
const handleCredentialInput = () => {
  const savedUsername = localStorage.getItem('username');
  const savedPassword = localStorage.getItem('password');

  if (savedUsername && savedPassword) {
    setUsername(savedUsername);
    setPassword(savedPassword);
  }
};
```

```
// App.tsx
```

```
const checkStoredCredentials = () => {
  const storedUsername = localStorage.getItem('username');
  const storedPassword = localStorage.getItem('password');
  if (storedUsername && storedPassword) {
    handleLogin(storedUsername, storedPassword);
  }
};
```

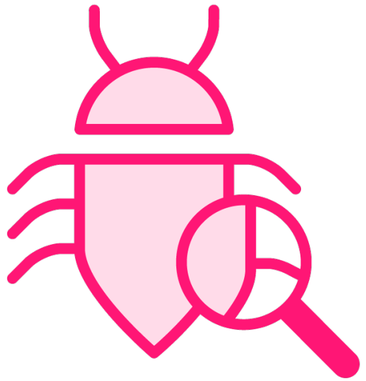
```
useEffect(() => {
  checkStoredCredentials();
}, []);
```

```
  const handleLogin = async (username: any, password: any) => {
    try {
      const response = await fetch('http://localhost:4000/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ username, password }),
      });

      if (response.ok) {
        localStorage.setItem('username', username);
        localStorage.setItem('password', password);

        setIsLoggedIn(true);
      }
       else {
        console.error('Login failed');
      }
    } catch (error) {
      console.error('Login error:', error);
    }
  };
```

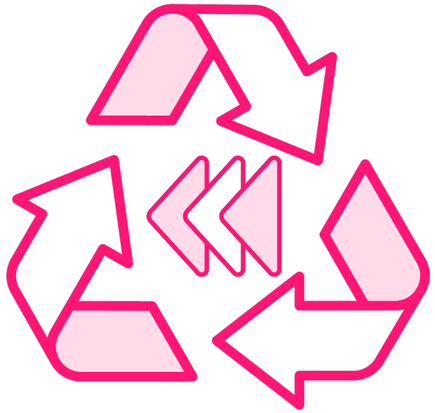
Client-Side Caching

Error Handling and Retry Strategies

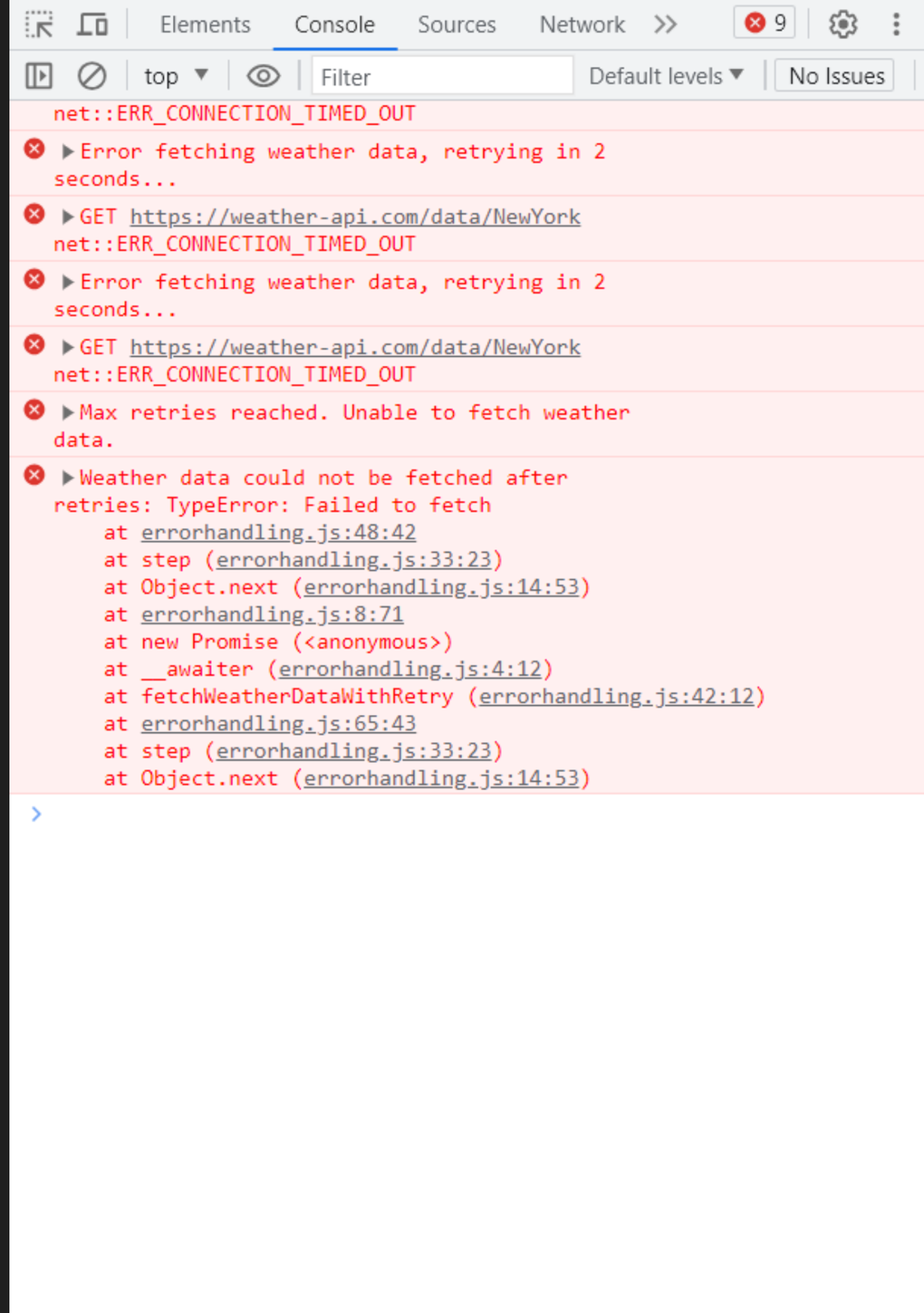
Error Handling



Retry Strategies



```typescript
const MAX_RETRIES = 3;
const RETRY_DELAY_MS = 2000;

async function fetchWeatherDataWithRetry(city: string, retries = MAX_RETRIES): Promise<any> {
  try {
    const response = await fetch(`https://weather-api.com/data/${city}`);

    if (!response.ok) {
      throw new Error('Failed to fetch weather data');
    }

    const data = await response.json();
    return data;
  } catch (error) {
    if (retries > 0) {
      console.error('Error fetching weather data, retrying in 2 seconds...');
      await new Promise(resolve => setTimeout(resolve, RETRY_DELAY_MS));
      return fetchWeatherDataWithRetry(city, retries - 1);
    } else {
      console.error('Max retries reached. Unable to fetch weather data.');
      throw error;
    }
  }
}

async function main() {
  const city = 'NewYork';
  try {
    const weatherData = await fetchWeatherDataWithRetry(city);


    console.log('Weather:', weatherData);
  } catch (error) {

    console.error('Weather data could not be fetched after retries:', error);
  }
}


main();
```



Efficient Data Fetching Techniques

Caching and Data Management

Pagination

Batching

Prefetching

Lazy Loading

Additional Considerations

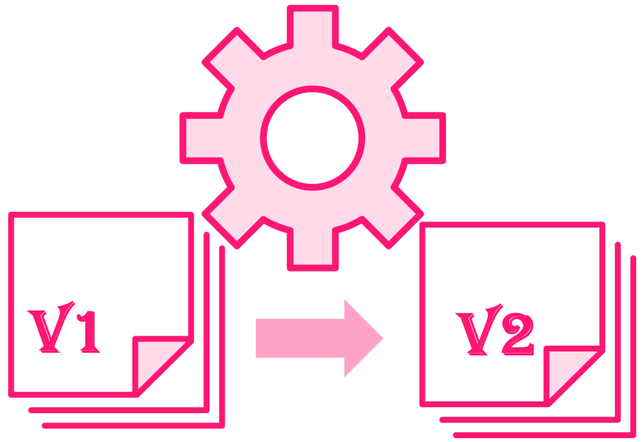
Selecting the Right Technique

Error Handling

Monitoring Tools

API Versioning and Backward Compatibility

API Versioning

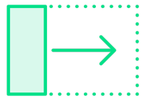


Importance of API Versioning

Preserving Compatibility

Avoiding Disruptions

Stakeholder Confidence



Flexibility

Backward Compatibility

```ts
interface UserV1
{
  id: number;
  name: string;
}


interface UserV2 {
  id: number;
  name: string;
  email: string;
}
```

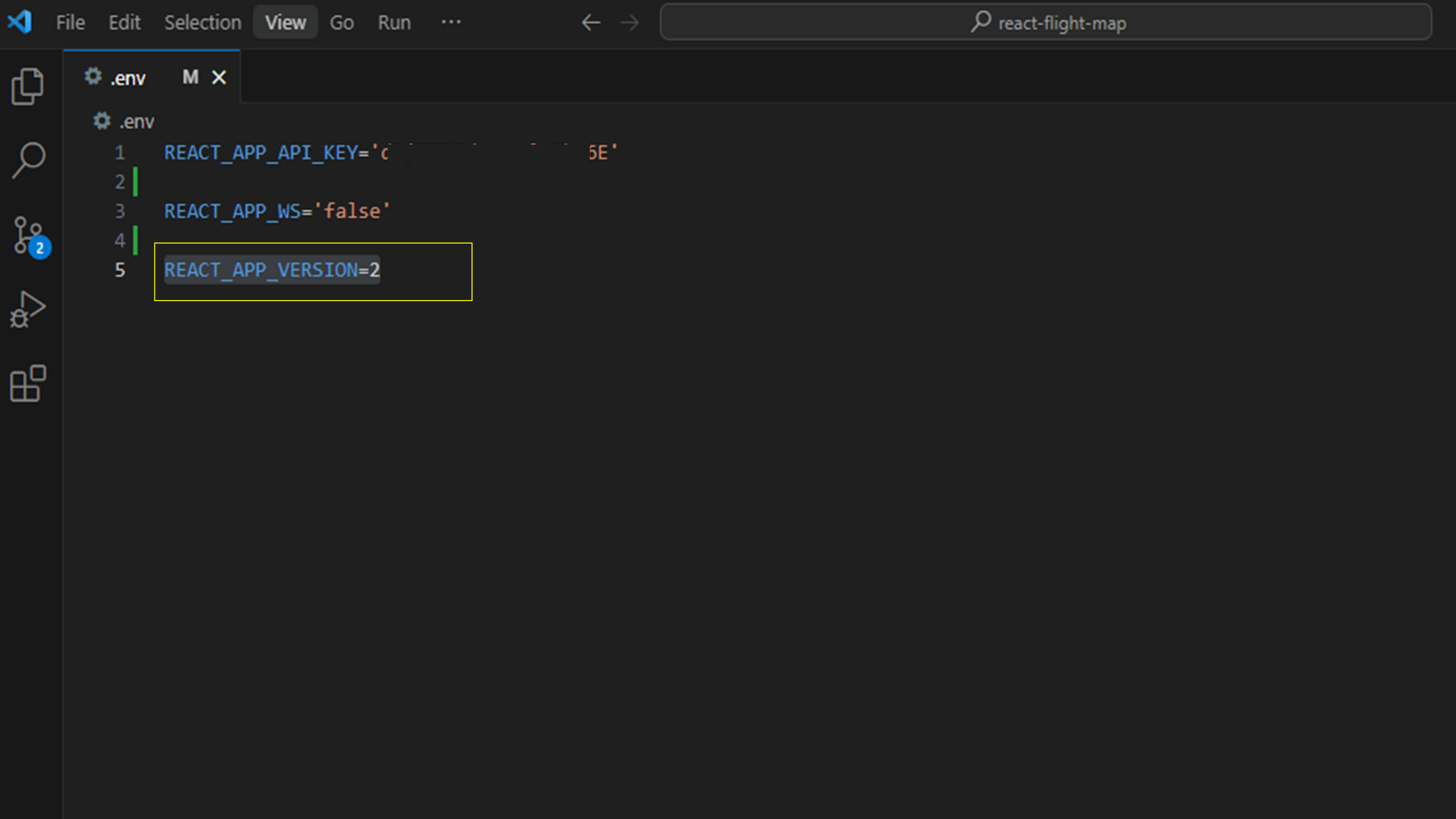
```ts
class UserClientV1
{
  constructor(private readonly baseUrl: string) {}

  async getUsers(): Promise<UserV1[]> {
    const response = await fetch('https://swapi.dev/api/people/');
    const users = await response.json();

    return users;
  }
}

class UserClientV2 {
  constructor(private readonly baseUrl: string) {}

  async getUsers(): Promise<UserV2[]> {
    const response = await fetch('https://swapi.dev/api/people/');
    const users = await response.json();

    return users;
  }
}
```

```
async function main()
{
  const userClientV1 = new UserClientV1('https://swapi.dev/api/people/');
  const userClientV2 = new UserClientV2('https://swapi.dev/api/people/');


  const usersV1 = await userClientV1.getUsers();
  console.log('Users V1:', usersV1);


  const usersV2 = await userClientV2.getUsers();
  console.log('Users V2:', usersV2);
}

main();
```

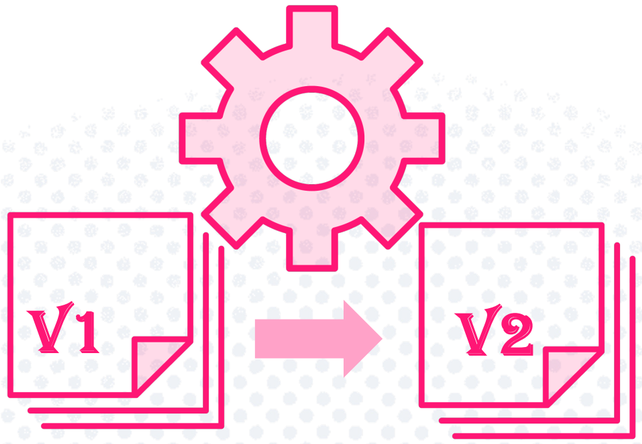
Practical Implementation

```
const v1Router = express.Router();
v1Router.get('/opensky-local', authenticate, (req, res) => {
  if (req.query.fileId) {
    let jsonFile = req.query.fileId;
    console.log(jsonFile);
    const filePath = `./uploads/${jsonFile}.json`;

    if (fs.existsSync(filePath)) {
      try {
        const dataFromFile = fs.readFileSync(filePath, 'utf-8');
        const jsonData = JSON.parse(dataFromFile);

        res.json({
          data: jsonData,
        });
      }
     catch (error) {
        console.error('Error parsing JSON data:', error);
        res.status(500).json({ error: 'Internal Server Error' });
      }
    } else {
      res.status(404).json({ error: 'File not found' });
    }
  } else {
    res.status(400).json({ error: 'Bad Request' });
  }
});
```

```
const v2Router = express.Router();
v2Router.get('/opensky-local', authenticate, (req, res) => {
  if (req.query.fileId) {
    let jsonFile = req.query.fileId;
    console.log(jsonFile);
    const filePath = `./uploads/${jsonFile}.json`;

    if (fs.existsSync(filePath)) {
      try {
        const dataFromFile = fs.readFileSync(filePath, 'utf-8');
        const jsonData = JSON.parse(dataFromFile);

        // Calculate the total number of states
        const totalStates = jsonData.states.length;

        res.json({
          data: jsonData,
          total: totalStates,
        });
      } catch (error) {
        console.error('Error parsing JSON data:', error);
        res.status(500).json({ error: 'Internal Server Error' });
      }
    } else {
      res.status(404).json({ error: 'File not found' });
    }
  } else {
    res.status(400).json({ error: 'Bad Request' });
  }
});

app.use('/v1', v1Router);
app.use('/v2', v2Router);
```

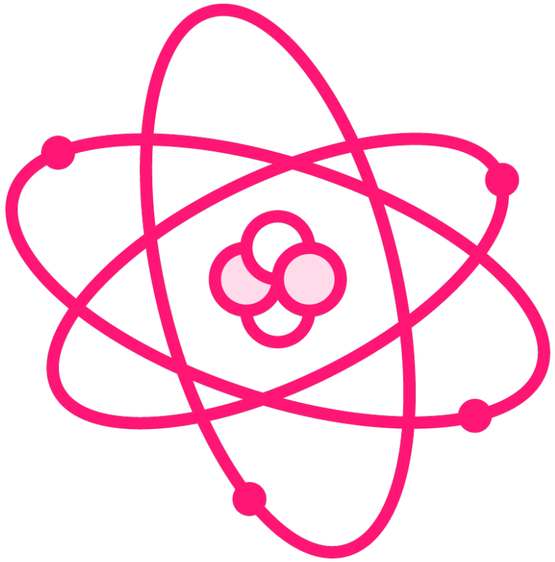

File
Edit
Selection
View
Go
Run
react-flight-map
.env
M
.env
1 REACT_APP_API_KEY='d 5E'
2
3 REACT_APP_WS='false'
4
5 REACT_APP_VERSION=2

API Versioning



Handling Concurrent Requests and Rate Limiting

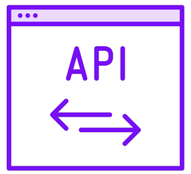
Concurrent Requests



Rate Limiting

Importance of Concurrent Requests and Rate Limiting

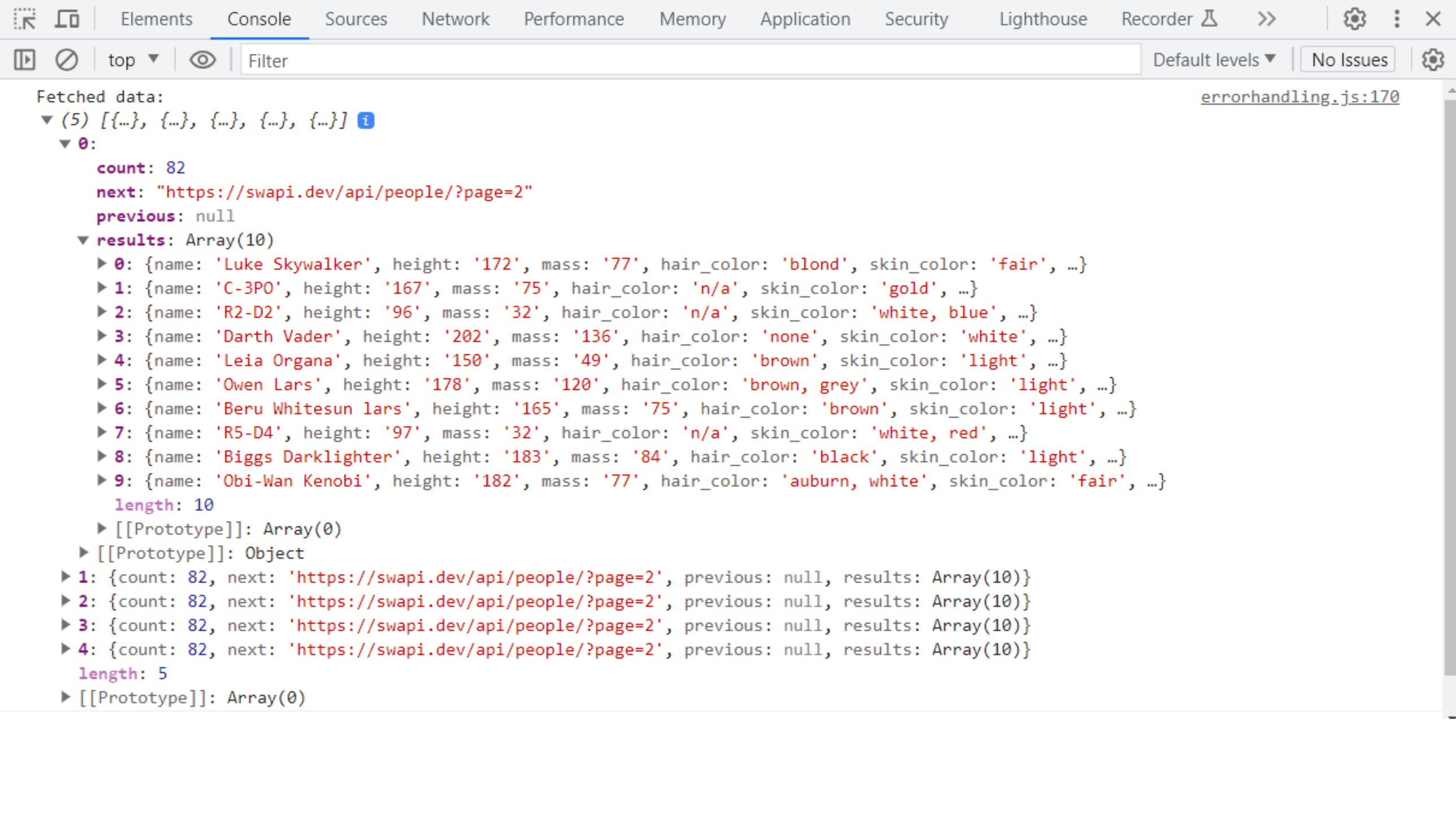
Performance Boost



API Protection

Resource Conservation

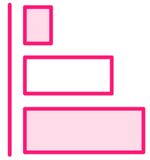
```
async function fetchDataWithRateLimit(url: string, limit: number): Promise<any[]> {
    const responses: Promise<any>[] = [];

    for (let i = 0; i < limit; i++) {
      responses.push(fetch(url));
    }

    try {
      const data = await Promise.all(responses);
      const parsedData = await Promise.all(data.map((response) => response.json()));
      return parsedData;
    } catch (error) {
      console.error('Error fetching data:', error);
      throw error;
    }
  }

  const apiUrl = 'https://swapi.dev/api/people/';
  const concurrentRequestLimit = 5;

  fetchDataWithRateLimit(apiUrl, concurrentRequestLimit)
    .then((data) => {
      console.log('Fetched data:', data);
    })
    .catch((error) => {
      console.error('Failed to fetch data:', error);
    });
```


Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
top
Filter
Default levels
No Issues
Fetched data:
errorhandling.js:170
(5) [{…}, {…}, {…}, {…}, {…}]
0:
count: 82
next: "https://swapi.dev/api/people/?page=2"
previous: null
results: Array(10)
0: {name: 'Luke Skywalker', height: '172', mass: '77', hair_color: 'blond', skin_color: 'fair', …}
1: {name: 'C-3PO', height: '167', mass: '75', hair_color: 'n/a', skin_color: 'gold', …}
2: {name: 'R2-D2', height: '96', mass: '32', hair_color: 'n/a', skin_color: 'white, blue', …}
3: {name: 'Darth Vader', height: '202', mass: '136', hair_color: 'none', skin_color: 'white', …}
4: {name: 'Leia Organa', height: '150', mass: '49', hair_color: 'brown', skin_color: 'light', …}
5: {name: 'Owen Lars', height: '178', mass: '120', hair_color: 'brown, grey', skin_color: 'light', …}
6: {name: 'Beru Whitesun lars', height: '165', mass: '75', hair_color: 'brown', skin_color: 'light', …}
7: {name: 'R5-D4', height: '97', mass: '32', hair_color: 'n/a', skin_color: 'white, red', …}
8: {name: 'Biggs Darklighter', height: '183', mass: '84', hair_color: 'black', skin_color: 'light', …}
9: {name: 'Obi-Wan Kenobi', height: '182', mass: '77', hair_color: 'auburn, white', skin_color: 'fair', …}
length: 10
[[Prototype]]: Array(0)
[[Prototype]]: Object
1: {count: 82, next: 'https://swapi.dev/api/people/?page=2', previous: null, results: Array(10)}
2: {count: 82, next: 'https://swapi.dev/api/people/?page=2', previous: null, results: Array(10)}
3: {count: 82, next: 'https://swapi.dev/api/people/?page=2', previous: null, results: Array(10)}
4: {count: 82, next: 'https://swapi.dev/api/people/?page=2', previous: null, results: Array(10)}
length: 5
[[Prototype]]: Array(0)

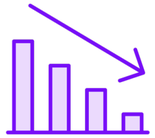
Implementing Long-Polling and Real-Time APIs

Importance of Long-Polling and Real-Time APIs

Real-Time Updates



Enhanced Interactivity



Reduced Latency

Resource Efficiency

```
import { Express } from 'express';

const app = Express();

app.get('/long-polling', async (req, res) => {

  const data = await new Promise((resolve, reject) => {

  });


  res.send(data);
});

app.listen(3000);
```

Implementing Long-Polling and Real-Time APIs

WebSockets

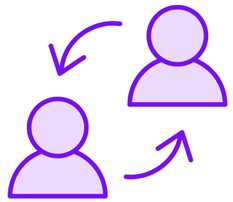
SSE (Server-Sent Events)

Comet

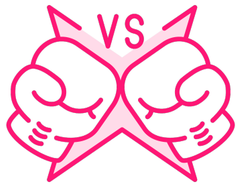
Long-Polling HTTP

Web Sockets and Server Sent Events (SSE)

Web Sockets



Full-Duplex Communication

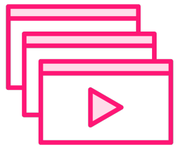


Contrast with Traditional HTTP

Ideal for applications that require instant updates or interactive features

Server-Sent Events (SSE)

Unidirectional Communication



Efficient Streaming

Useful when the server needs to push real-time updates to clients

Importance of Web Sockets and SSE

Real-Time Updates

Reduced Latency

Efficiency

Interactive Features

```
const socket = new WebSocket('wss://socketsbay.com/wss/v2/1/demo/');
```

```
socket.addEventListener('open', (event) => {
  console.log('WebSocket connection opened:', event);

  socket.send('Hello, WebSocket server!');
});
```

```
socket.addEventListener('message', (event) => {
  console.log('Message from server:', event.data);
});
```

```
socket.addEventListener('error', (event) => {
  console.error('WebSocket error:', event);
});
```

```
socket.addEventListener('close', (event) => {
  if (event.wasClean) {
    console.log('WebSocket connection closed cleanly:', event);
  } else {
    console.error('WebSocket connection abruptly closed:', event);
  }
});
```

```
const eventSource = new EventSource('https://example.com/sse-endpoint');
```

```
eventSource.addEventListener('message', (event) => {

  const data = JSON.parse(event.data);
  console.log('Received SSE message:', data);


});
```

```
eventSource.addEventListener('open', (event) => {

  console.log('SSE connection opened:', event);
});
```

```
eventSource.addEventListener('error', (error) => {

  console.error('SSE error:', error);


  if (eventSource.readyState === EventSource.CLOSED) {
    console.log('SSE connection closed.');
  }
});
```

```
// Close the SSE connection when needed
// eventSource.close();
```

Practical Implementation

- Handling Concurrent Requests and Rate Limiting
- Implementing Long-Polling
- Web Sockets and Server-Sent Events (SSE)
- Complex Authorization Scenarios

```
// Implementing Rate Limiting
import { RateLimiter } from "limiter";
```

```
// Allow 150 requests per hour.
const limiter = new RateLimiter({ tokensPerInterval: 150, interval: "hour" });
```

```
// Add await apiLimiter.removeTokens(1); into the GET method
v2Router.get('/opensky-local', authenticate, async (req, res) => {
  try {
    await apiLimiter.removeTokens(1);
    // ... Request handling logic
  } catch (error) {
    res.status(429).json({ error: 'Rate limit exceeded' });
  }
});
```

```
// Implementing Rate Limiting
import { RateLimiter } from "limiter";

// Allow 150 requests per hour.
const limiter = new RateLimiter({ tokensPerInterval: 150, interval: "hour" });

// Add await apiLimiter.removeTokens(1); into the GET method
v2Router.get('/opensky-local', authenticate, async (req, res) => {
  try {
    await apiLimiter.removeTokens(1);
    // ... Request handling logic
  }

  catch (error) {
    res.status(429).json({ error: 'Rate limit exceeded' });
  }

});
```

```
// Implementing WebSockets
const WebSocket = require('ws');
const wss = new WebSocket.Server({ server });

// WebSocket Server Logic
// ... (WebSocket server logic)
```

```
// WebSocket Client Integration
const webSocket = process.env.REACT_APP_WS;
```

```
// map.tsx

  useEffect(() => {
    if (webSocket === 'true') {
      const wsUrl = 'ws://localhost:4000';
      const ws = new WebSocket(wsUrl);

      ws.onopen = () => {
        console.log('WebSocket connection established');
      };

      ws.onmessage = (event) => {
        try {
          const data = JSON.parse(event.data);
          setFlightData(data);
        } catch (error) {
          console.error('Error parsing WebSocket data:', error);
        }
      };
```

```
import { RateLimiter } from "limiter";


const limiter = new RateLimiter({ tokensPerInterval: 150, interval: "hour" });

v2Router.get('/opensky-local', authenticate, async (req, res) => {
  try {
    await apiLimiter.removeTokens(1);
    if (req.query.fileId) {
      let jsonFile = req.query.fileId;
      console.log(jsonFile);
      const filePath = `./uploads/${jsonFile}.json`;

      if (fs.existsSync(filePath)) {
        try {
          const dataFromFile = fs.readFileSync(filePath, 'utf-8');
          const jsonData = JSON.parse(dataFromFile);

          const totalStates = jsonData.states.length;

          res.json({
            data: jsonData,
            total: totalStates,
          });
        } catch (error) {
          console.error('Error parsing JSON data:', error);
          res.status(500).json({ error: 'Internal Server Error' });
        }
      }
```

```
else {
        res.status(404).json({ error: 'File not found' });
      }
    } else {
      res.status(400).json({ error: 'Bad Request' });
    }
  } catch (error) {
    res.status(429).json({ error: 'Rate limit exceeded' });
  }
});

const WebSocket = require('ws');
const wss = new WebSocket.Server({ server });

let counter = 0;

let broadcastInterval;

function broadcastDataToClients() {
  const fileName = `data${counter}.json`;
  const filePath = `./uploads/${fileName}`;

  if (fs.existsSync(filePath)) {
    const dataFromFile = fs.readFileSync(filePath, 'utf-8');

    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        try {
          const jsonData = JSON.parse(dataFromFile);
          client.send(JSON.stringify(jsonData));
        } catch (error) {
          console.error('Error parsing JSON data:', error);
        }
      }
    });

    counter++;
  } else {
    console.log(`File ${fileName} not found. Broadcasting stopped.`);
    clearInterval(broadcastInterval);
  }
}
```

```
broadcastInterval = setInterval(broadcastDataToClients, 3000);

wss.on('connection', (ws) => {
  console.log('WebSocket client connected');

  ws.on('message', (message) => {
    console.log(`Received message: ${message}`);
  });

  ws.on('close', () => {
    console.log('WebSocket client disconnected');
  });
});

const webSocket = process.env.REACT_APP_WS;

useEffect(() => {
  if (webSocket === 'true') {
    const wsUrl = 'ws://localhost:4000';
    const ws = new WebSocket(wsUrl);

    ws.onopen = () => {
      console.log('WebSocket connection established');
    };

    ws.onmessage = (event) => {
      try {
        const data = JSON.parse(event.data);
        setFlightData(data);
      } catch (error) {
        console.error('Error parsing WebSocket data:', error);
      }
    };
```

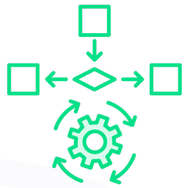
```
    ws.onclose = () => {
      console.log('WebSocket connection closed');
    };

    return () => {
      ws.close();
    };
  }
}, [webSocket]);
```

Exploring Emerging API Technologies

API-as-a-Product Revolution

Serverless Architecture Simplifies API Consumption

GraphQL's Flexibility and Efficiency

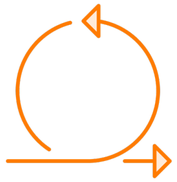


OpenAPI Specification for Reduced Boilerplate Code

Strategies for Future Proofing

Stay Informed with Release Notes

Linting and Code Style Guides



Version Control and Branching

Documentation