Fetch API Assignment - Complete Documentation

Student Name: [Priyansh Pal]
Assignment: Daily Assignment on FETCH API
Course: Wipro NGA MERN - Week 1
Submission Date: [31-10-2025]

Assignment Objective

The objective of this assignment was to develop a blogging platform with todo list management functionality, demonstrating proficiency in:

- **Fetch API** for asynchronous data retrieval
- **JSON data handling** and parsing
- **DOM manipulation** for dynamic content rendering
- **Error handling** mechanisms in JavaScript
- **JavaScript design patterns** for code organization
- **API integration** with JSONPlaceholder public APIs

Technical Approach

1. Technology Stack

- **HTML5**: Semantic markup for structure
- **CSS3**: Modern styling with Flexbox and Grid
- **Vanilla JavaScript (ES6+)**: Core functionality implementation
- **Fetch API**: Asynchronous HTTP requests
- **JSONPlaceholder API**: Remote data source

2. Design Pattern Selection

I chose the **Revealing Module Pattern** for this project because:

- **Encapsulation**: Private variables and functions are truly private
- **Clean Public API**: Only necessary functions are exposed
- **Namespace Management**: Avoids global scope pollution
- **Maintainability**: Clear separation of concerns

- **Scalability**: Easy to extend functionality

Implementation Details

Task 1: Fetch and Display Posts

Objective: Use Fetch API to retrieve and display blog posts

Implementation Steps:

1. **API Call Function:**

```
async function fetchDataFromAPI(endpoint) {
  const url = `${API_CONFIG.BASE_URL}${endpoint}`;
  const response = await fetch(url);

  // Validate response status
  if (!response.ok) {
    throw new Error(`HTTP ${response.status}: ${response.statusText}`);
  }

  // Parse and return JSON data
  return await response.json();
}
```

2. **Rendering Logic:**
    - Fetched posts from /posts endpoint
    - Limited display to 15 posts for performance
    - Used Array.map() to generate HTML for each post
    - Implemented HTML escaping to prevent XSS attacks
    - Added post metadata (User ID, Post ID)
3. **DOM Manipulation:**
    - Created dynamic post cards with proper structure
    - Applied CSS classes for styling

- Used innerHTML for efficient bulk insertion

Key Features:

- Asynchronous data fetching with async/await
- Dynamic HTML generation
- Security: XSS prevention via HTML escaping
- User-friendly post card design
- Scrollable container for better UX

Task 2: Fetch and Display Todos

Objective: Use Fetch API to retrieve and display todo list items

Implementation Steps:

1. **API Integration:**
    - Fetched todos from /todos endpoint
    - Reused generic fetchDataFromAPI() function
    - Limited display to 25 todos
2. **Data Processing:**

```
function calculateTodoStats(todos) {
  const total = todos.length;
  const completed = todos.filter(todo => todo.completed).length;
  const pending = total - completed;

  return { total, completed, pending };
}
```

3. **Visual Representation:**
    - Checkbox indicators for completion status
    - Color-coded completed vs pending tasks
    - Strike-through text for completed todos
    - Statistics dashboard showing total/completed/pending

Key Features:

- Status-based visual differentiation
- Real-time statistics calculation
- Interactive todo item design
- Completion percentage tracking

Task 3: Error Handling

Objective: Implement comprehensive error handling mechanisms

Error Types Handled:

1. **Network Errors:**

```
if (error.name === 'TypeError' && error.message.includes('fetch')) {
  throw new Error(
    'Network Error: Unable to connect to the server. ' +
    'Please check your internet connection and try again.'
  );
}
```

2. **HTTP Errors:**
    - Status code validation (checking response.ok)
    - Custom error messages for different status codes
    - User-friendly error descriptions
3. **Data Validation Errors:**
    - Content-type validation
    - Empty data checking
    - Array structure validation
4. **JSON Parsing Errors:**
    - SyntaxError handling
    - Invalid data format detection

Error Display Strategy:

- Visually distinct error cards (red background)
- Clear error titles and detailed messages
- Console logging for developer debugging
- Refresh buttons to retry failed operations

Example Error Handling:

```
try {
  const posts = await fetchDataFromAPI(API_CONFIG.ENDPOINTS.POSTS);
  renderPosts(posts);
} catch (error) {
  showErrorState(
    container,
    'Failed to Load Blog Posts',
    error.message
  );
}
```

Task 4: JavaScript Design Patterns

Pattern Used: Revealing Module Pattern

Why This Pattern?

The Revealing Module Pattern provides:

1. **Privacy:** True private variables and functions
2. **Organization:** Clear code structure
3. **Maintainability:** Easy to understand and modify
4. **Reusability:** Modular function design
5. **Testing:** Public API is easily testable

Implementation Structure:

```
const BlogPlatform = (function() {
  // PRIVATE SECTION
```

```
  const API_CONFIG = { /* private config */ };

  function privateFunction() {
    // Only accessible within module
  }

  // PUBLIC SECTION
  return {
    publicFunction: publicFunction,
    init: initializeApplication
  };
})(); // IIFE - Immediately Invoked Function Expression
```

Benefits Demonstrated:

- No global variable pollution
- Clear public API interface
- Encapsulated business logic
- Single responsibility principle
- Easy to extend and maintain

Task 5: Integration and Testing

Integration Approach:

1. **Concurrent Loading:**

```
await Promise.allSettled([
  fetchAndDisplayPosts(),
  fetchAndDisplayTodos()
]);
```

2. **Initialization Sequence:**
    - Wait for DOM ready event

- Initialize DOM references
- Load posts and todos in parallel
- Display success/error states

Challenges Faced

Challenge 1: Asynchronous Data Loading

Problem:
Managing multiple asynchronous API calls and ensuring proper error handling for each.

Solution:

- Used Promise.allSettled() instead of Promise.all() to prevent one failure from stopping all operations
- Implemented individual try-catch blocks for each API call
- Added loading states to provide user feedback

Code Example:

```
await Promise.allSettled([
  fetchAndDisplayPosts(),
  fetchAndDisplayTodos()
]);
// Both execute independently, failures don't cascade
```

Challenge 2: Error Type Differentiation

Problem:
Different types of errors (network, HTTP, parsing) needed specific user messages.

Solution:

- Implemented error type detection based on error.name and message patterns
- Created custom error messages for each scenario
- Added fallback generic error handler

Implementation:

```
if (error.name === 'TypeError') {
```

```
  // Network error
} else if (error.name === 'SyntaxError') {
  // JSON parsing error
} else if (!response.ok) {
  // HTTP error
}
```

Challenge 3: XSS Security Concerns

Problem:
Rendering user-generated content (titles, bodies) could expose XSS vulnerabilities.

Solution:

- Created escapeHTML() function to sanitize all user content
- Used textContent property for secure text insertion
- Validated and escaped all dynamic content before rendering

Security Function:

```
function escapeHTML(text) {
  const div = document.createElement('div');
  div.textContent = text;  // Automatically escapes HTML
  return div.innerHTML;
}
```

Challenge 4: Design Pattern Implementation

Problem:
Choosing and correctly implementing a JavaScript design pattern while maintaining code readability.

Solution:

- Selected Revealing Module Pattern for its balance of encapsulation and simplicity
- Organized code into clear sections (private/public)

- Added extensive comments for clarity
- Used IIFE to create true privacy

Challenge 5: Responsive Design

Problem:
Ensuring the application works well on both desktop and mobile devices.

Solution:

- Implemented CSS Grid with responsive breakpoints
- Used @media queries for mobile optimization
- Created scrollable containers for long lists
- Tested on multiple viewport sizes

CSS Implementation:

```
.content-wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 30px;
}

@media (max-width: 968px) {
  .content-wrapper {
    grid-template-columns: 1fr;
  }
}
```

Challenge 6: Performance Optimization

Problem:
Loading and rendering 100+ posts and 200+ todos could cause performance issues.

Solution:

- Implemented data limiting (15 posts, 25 todos)

- Used document fragments for batch DOM updates
- Cached DOM references to avoid repeated queries
- Optimized rendering with single innerHTML assignment

Console Logging:

Implemented comprehensive logging for debugging:

Initializing Blog Platform Application

Fetching data from: https://jsonplaceholder.typicode.com/posts

Successfully fetched 100 items

Rendered 15 posts to DOM

Task 1 completed: Posts fetched and displayed successfully

Learning Reflection

Key Learnings:

1. **Fetch API Mastery:**
    - Understood the promise-based nature of Fetch API
    - Learned proper async/await syntax and error handling
    - Discovered the importance of response validation
    - Mastered JSON parsing and data transformation
2. **Error Handling Best Practices:**
    - Realized the importance of user-friendly error messages
    - Learned to differentiate between error types
    - Understood the need for graceful degradation
    - Discovered the value of logging for debugging
3. **Design Patterns:**
    - Appreciated the benefits of code organization
    - Understood encapsulation and privacy in JavaScript
    - Learned IIFE pattern for module creation

    - Recognized the importance of public API design
4. **DOM Manipulation:**
    - Mastered dynamic content generation
    - Learned efficient bulk DOM updates
    - Understood the importance of caching DOM references
    - Discovered performance optimization techniques
5. **Security Awareness:**
    - Learned about XSS vulnerabilities
    - Understood the importance of input sanitization
    - Discovered HTML escaping techniques
    - Recognized security as a fundamental requirement

Insights Gained:

1. **Code Organization Matters:**
   Well-organized code is easier to debug, maintain, and extend. The Revealing Module Pattern made my code significantly more manageable.
2. **User Experience is Critical:**
   Loading states, error messages, and visual feedback dramatically improve user experience. Users should always know what's happening.
3. **Error Handling is Not Optional:**
   Robust error handling is essential for production applications. Users need helpful messages, not cryptic errors.
4. **Performance Considerations:**
   Even with small datasets, performance optimization (limiting data, caching references) creates a better experience.
5. **Real-world API Integration:**
   Working with external APIs requires handling various edge cases, validation, and error scenarios that may not be obvious initially.

Skills Developed:

- Asynchronous JavaScript (Promises, async/await)
- RESTful API consumption
- JSON data handling and transformation

- Advanced DOM manipulation
- Error handling and debugging
- JavaScript design patterns
- Code organization and architecture
- Security best practices
- Responsive web design
- Performance optimization

Project Achievements:

- Successfully integrated with JSONPlaceholder API
- Implemented all 5 tasks as specified
- Created a production-ready, scalable application
- Demonstrated understanding of modern JavaScript practices
- Delivered a user-friendly, responsive interface

Personal Growth:

This assignment has significantly enhanced my understanding of:

- Asynchronous programming patterns in JavaScript
- API integration and data management
- Error handling strategies
- Code organization and architecture
- Frontend development best practices

Appendix

API Endpoints Used:

- **Posts API:** https://jsonplaceholder.typicode.com/posts
- **Todos API:** https://jsonplaceholder.typicode.com/todos

Browser Console Commands for Testing:

// Manually trigger posts refresh

```
BlogPlatform.fetchAndDisplayPosts();

// Manually trigger todos refresh
BlogPlatform.fetchAndDisplayTodos();

// Reinitialize application
BlogPlatform.init();
```

Student Name: [Priyansh Pal]
Assignment: Daily Assignment on FETCH API
Course: Wipro NGA MERN - Week 1
Submission Date: [31-10-2025]