Web-Entwicklung

Jil Zerndt, Lucien Perret
January 2025

Web Development Introduction

Web Architecture **Client-Server Model:**
- Browser (Client) sendet Anfragen an Server
- Server verarbeitet Anfragen und sendet Antworten
- Kommunikation über HTTP/HTTPS (Port 80/443)

Core Technologies **Client-Side** → Front-end Development
- HTML: Structure and content
- CSS: Styling and layout
- JavaScript: Behavior and interactivity
- Browser APIs and Web Standards

Server-Side → Back-end Development
- Choice of platform and programming language
- Generates browser-compatible output
- Examples: Node.js, Express, REST APIs

Internet vs. WWW **Internet:**
- Global network of interconnected computer networks
- Various services: Email, FTP, WWW, etc.
- Core protocols: TCP/IP
- Originally ARPANET (1969)

World Wide Web:
- Service built on top of the Internet
- Developed by Tim Berners-Lee at CERN (1990s)
- Based on: HTTP, HTML, URLs
- Browser as client application

Web Standards **Standards Organizations:**
- **W3C** (World Wide Web Consortium)
  - Founded 1994 at MIT
  - Led by Tim Berners-Lee
  - Standardizes web technologies
- **WHATWG** (Web Hypertext Application Technology Working Group)
  - Founded by Apple, Mozilla, Opera
  - Later joined by Microsoft, Google
  - Maintains HTML Living Standard
- **Browser Vendors**
  - Implement and influence standards
  - Chrome, Firefox, Safari, Edge
  - Growing influence on web development

Common Ports

Port	Service
20	FTP (Data)
21	FTP (Control)
22	SSH
23	Telnet
25	SMTP
53	DNS
80	HTTP
443	HTTPS

Web Development Approaches Historical evolution:
1. Static web pages
2. Server-generated content (CGI, Perl)
3. Server-side scripting (PHP)
4. Client-side scripting (JavaScript)
5. Single Page Applications (SPAs)
6. Modern web frameworks

Current trends:
- Component-based development
- Client-side rendering
- RESTful APIs
- Progressive Web Apps (PWAs)
- Responsive design

JavaScript

JavaScript Overview
- Created by Brendan Eich in 1995 for Netscape Navigator
- Dynamic, weakly typed programming language
- Multi-paradigm: Object-oriented, functional, imperative
- Originally for client-side scripting, now also server-side (Node.js)
- Regular updates via ECMAScript standard

Core Language Features

Data Types Primitive Types:
- `number`: 64-bit floating point (IEEE 754)
- `bigint`: arbitrary precision integers (with n suffix)
- `string`: text in ”, , or “
- `boolean`: true/false
- `undefined`: uninitialized value
- `null`: intentionally empty value
- `symbol`: unique identifier

Type Checking

```
// Type checking with typeof
typeof 42           // 'number'
typeof 42n          // 'bigint'
typeof "text"       // 'string'
typeof true         // 'boolean'
typeof undefined    // 'undefined'
typeof null         // 'object' (historical bug!)
typeof {}           // 'object'
typeof []           // 'object'
typeof (() => {})   // 'function'

// Special number values
console.log(Infinity)    // Division by zero
console.log(NaN)        // Invalid numeric operation
```

Variables Three ways to declare variables:
- `var`: function-scoped, hoisted (avoid)
- `let`: block-scoped, mutable
- `const`: block-scoped, immutable reference

Control Structures

```
// Conditionals
if (condition) {
    // code
} else if (otherCondition) {
    // code
} else {
    // code
}

// Switch statement
switch(value) {
    case 1:
        // code
        break;
    default:
        // code
}

// Loops
for (let i = 0; i < n; i++) { }
while (condition) { }
do { } while (condition);
for (let item of array) { }
for (let key in object) { }
```

Objects and Arrays

Objects Key characteristics:
- Collections of key-value pairs
- Dynamic - properties can be added/removed
- Keys are strings or symbols
- Values can be any type, including functions (methods)
- Prototype-based inheritance

Object Manipulation

```
// Object creation
const person = {
    name: "Alice",
    age: 30,
    greet() {
        return `Hello, I'm ${this.name}`;
    }
};

// Property access
person.name             // dot notation
person["age"]           // bracket notation

// Property manipulation
person.job = "Developer";     // add
delete person.age;            // delete
"name" in person;            // check existence

// Object methods
Object.keys(person)          // get keys
Object.values(person)        // get values
Object.entries(person)       // get key-value pairs
Object.assign(target, ...sources)  // merge objects
```

Arrays Special objects for ordered collections:

- Zero-based indexing
- Dynamic length
- Can contain mixed types
- Many built-in methods for manipulation

Array Methods

```
const arr = [1, 2, 3];

// Modifying arrays
arr.push(4);              // add to end
arr.pop();               // remove from end
arr.unshift(0);          // add to start
arr.shift();             // remove from start
arr.splice(1, 1, 'new'); // remove/insert at position

// Accessing arrays
arr.slice(1, 3);         // get sub-array
arr.indexOf(2);          // find element
arr.includes(2);         // check existence

// Functional methods
arr.map(x => x * 2);     // transform elements
arr.filter(x => x > 2); // filter elements
arr.reduce((a, b) => a + b); // reduce to value
arr.forEach(x => console.log(x)); // iterate
```

Functions and Closures

Functions Functions in JavaScript are first-class objects:

- Can be assigned to variables
- Passed as arguments
- Returned from other functions
- Have their own properties and methods

Function Declarations

```
// Function declaration
function greet(name) {
    return `Hello, ${name}!`;
}

// Function expression
const greet = function(name) {
    return `Hello, ${name}!`;
};

// Arrow function
const greet = name => `Hello, ${name}!`;

// Arrow function with multiple parameters
const add = (a, b) => a + b;

// Arrow function with block
const calculate = (a, b) => {
    const result = a * b;
    return result;
};
```

Function Parameters

```
// Default parameters
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}

// Rest parameters
function sum(...numbers) {
    return numbers.reduce((a, b) => a + b, 0);
}

// Destructuring parameters
function printPerson({name, age}) {
    console.log(`${name} is ${age} years old`);
}

// Spread operator
const numbers = [1, 2, 3];
console.log(Math.max(...numbers));
```

Closures A closure is created when a function is defined inside another function:

- Has access to variables in outer function scope
- Maintains access even after outer function returns
- Used for data privacy and state management

Closure Example

```
function createCounter() {
    let count = 0;
    return {
        increment() { return ++count; },
        decrement() { return --count; },
        getCount() { return count; }
    };
}

const counter = createCounter();
counter.increment(); // 1
counter.increment(); // 2
counter.decrement(); // 1
```

Asynchronous Programming

Asynchronous JavaScript Methods for handling asynchronous operations:

- Callbacks (traditional)
- Promises (modern)
- Async/Await (modern, cleaner syntax)

Promises

```
// Creating a Promise
const myPromise = new Promise((resolve, reject) => {
    // Async operation
    setTimeout(() => {
        if (success) {
            resolve('Operation completed');
        } else {
            reject('Operation failed');
        }
    }, 1000);
});

// Using a Promise
myPromise
    .then(result => console.log(result))
    .catch(error => console.error(error))
    .finally(() => console.log('Cleanup'));

// Async/Await
async function fetchData() {
    try {
        const result = await myPromise;
        console.log(result);
    } catch (error) {
        console.error(error);
    }
}
```

Node.js and Modules

Node.js Server-side JavaScript runtime:

- Built on Chrome's V8 engine
- Event-driven, non-blocking I/O
- Large ecosystem (npm)
- Used for web servers, CLI tools, etc.

Module Systems

```
// CommonJS (Node.js)
const fs = require('fs');
module.exports = { /* exports */ };

// ES Modules
import { function1 } from './module.js';
export const variable = 42;
export default class MyClass { /* ... */ }

// Package.json
{
    "name": "my-project",
    "version": "1.0.0",
    "dependencies": {
        "express": "^4.17.1"
    }
}
```

Browser Technologies

Document Object Model (DOM)

DOM Structure

- Tree representation of HTML document
- Each HTML element becomes a node
- Nodes can be elements, text, or attributes
- Provides API for dynamic manipulation
- Foundation for interactive web applications

DOM Manipulation

```
// Selecting elements
const element = document.getElementById('myId');
const elements =
    document.getElementsByClassName('myClass');
const element = document.querySelector('.myClass');
const elements =
    document.querySelectorAll('div.myClass');

// Creating elements
const div = document.createElement('div');
const text = document.createTextNode('Hello');
div.appendChild(text);

// Modifying elements
element.innerHTML = '<span>New content</span>';
element.textContent = 'New text';
element.setAttribute('class', 'newClass');
element.classList.add('newClass');
element.style.backgroundColor = 'red';

// Tree navigation
element.parentNode
element.childNodes
element.children
element.firstChild
element.nextSibling
```

Events

Event Handling

Events represent interactions or state changes:

- User interactions (clicks, keyboard input)
- Document loading stages
- Network status changes
- Timer completions

Event Listeners

```
// Adding event listeners
element.addEventListener('click', (event) => {
    console.log('Clicked!', event);
    event.preventDefault();  // Prevent default
        behavior
    event.stopPropagation(); // Stop event bubbling
});

// Removing event listeners
const handler = (event) => {
    console.log('Handler');
};
element.addEventListener('click', handler);
element.removeEventListener('click', handler);

// Event delegation
document.addEventListener('click', (event) => {
    if (event.target.matches('.button')) {
        // Handle button clicks
    }
});
```

Common Events

- Mouse: `click`, `dblclick`, `mouseover`, `mouseout`
- Keyboard: `keydown`, `keyup`, `keypress`
- Form: `submit`, `change`, `input`, `focus`, `blur`
- Document: `DOMContentLoaded`, `load`
- Window: `resize`, `scroll`

Browser APIs

Web APIs

Modern browsers provide numerous APIs:

- Storage (localStorage, sessionStorage)
- Fetch (network requests)
- Canvas and WebGL (graphics)
- Web Workers (parallel processing)
- Geolocation
- WebSockets (real-time communication)

Web Storage

```
// localStorage (persists between sessions)
localStorage.setItem('key', 'value');
const value = localStorage.getItem('key');
localStorage.removeItem('key');
localStorage.clear();

// sessionStorage (cleared when session ends)
sessionStorage.setItem('key', 'value');
const value = sessionStorage.getItem('key');

// Storing objects
const user = { name: 'John', age: 30 };
localStorage.setItem('user', JSON.stringify(user));
const storedUser =
    JSON.parse(localStorage.getItem('user'));
```

Fetch API

```
// GET request
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));

// POST request
fetch('https://api.example.com/data', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        name: 'John',
        age: 30
    })
})
.then(response => response.json())
.then(data => console.log(data));

// With async/await
async function fetchData() {
    try {
        const response = await
            fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}
```

Forms and HTTP

HTML Forms

Forms enable user input and data submission:

- `<form>` element with action and method
- Various input types (text, password, checkbox, etc.)
- Form validation (HTML5 and JavaScript)
- Data submission via GET or POST

Form Handling

```
// Form submission
const form = document.querySelector('form');
form.addEventListener('submit', async (event) => {
    event.preventDefault();

    const formData = new FormData(form);
    try {
        const response = await fetch('/submit', {
            method: 'POST',
            body: formData
        });
        const result = await response.json();
        console.log(result);
    } catch (error) {
        console.error('Error:', error);
    }
});

// Form validation
const input = document.querySelector('input');
input.addEventListener('input', (event) => {
    if (input.validity.typeMismatch) {
        input.setCustomValidity('Please enter a valid
            email');
    } else {
        input.setCustomValidity('');
    }
});
```

HTTP Methods

Method	Purpose
GET	Retrieve data
POST	Create new resource
PUT	Update entire resource
PATCH	Partial update
DELETE	Remove resource

Express.js

Express Framework Minimal web application framework for Node.js:
- Routing system
- Middleware support
- Static file serving
- Template engine integration
- Error handling

Express Basic Server

```
const express = require('express');
const app = express();

// Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
    res.send('Hello World');
});

app.post('/api/data', (req, res) => {
    const data = req.body;
    // Process data
    res.json({ success: true, data });
});

// Error handling
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});

// Start server
app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

Security Considerations

Web Security Common security concerns:
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- SQL Injection
- Session Hijacking
- Man-in-the-Middle Attacks

Security Best Practices

```
// Input sanitization
const sanitizeHTML = require('sanitize-html');
const cleanHTML = sanitizeHTML(dirtyHTML);

// CSRF Protection
app.use(csrf());
<form>
    <input type="hidden" name="_csrf" value="<%=
        csrfToken %>">
</form>

// Secure cookies
app.use(session({
    secret: 'secret-key',
    cookie: {
        secure: true,
        httpOnly: true,
        sameSite: 'strict'
    }
}));

// CORS
app.use(cors({
    origin: 'https://trusted-domain.com',
    methods: ['GET', 'POST']
}));
```

UI Libraries and Components

Modern Web Development

Component-Based Architecture

Key principles:

- Reusable, self-contained components
- Unidirectional data flow
- Declarative UI definition
- Virtual DOM for efficient updates
- Component lifecycle management

Framework vs Library

- **Library**
  - Collection of tools/functions
  - Application controls flow
  - Example: jQuery
- **Framework**
  - Provides application structure
  - Controls program flow
  - Example: Angular

JSX and SJDON

JSX Syntax

```
// JSX Component
const Welcome = ({name}) => (
    <div className="welcome">
        <h1>Hello, {name}</h1>
        <p>Welcome to our site!</p>
    </div>
);

// Nested Components
const App = () => (
    <div>
        <Welcome name="User" />
        <div className="content">
            <p>Main content here</p>
        </div>
    </div>
);
```

SJDON Syntax

```
// SJDON Component
const Welcome = ({name}) => [
    "div", {className: "welcome"},
    ["h1", `Hello, ${name}`],
    ["p", "Welcome to our site!"]
];

// Nested Components
const App = () => [
    "div",
    [Welcome, {name: "User"}],
    ["div", {className: "content"},
        ["p", "Main content here"]
    ]
];
```

Component State Management

State Management

State types:

- Local component state
- Shared/global state
- Props (passed from parent)
- Derived state (computed from other state)

State Hook Usage

```
const Counter = () => {
    // State declaration
    const [count, setCount] = useState(0);
    const [text, setText] = useState("");

    // Event handlers
    const increment = () => setCount(count + 1);
    const handleInput = (e) => setText(e.target.value);

    return [
        "div",
        ["h1", `Count: ${count}`],
        ["button", {onClick: increment}, "Increment"],
        ["input", {
            value: text,
            onInput: handleInput
        }]
    ];
};
```

Effect Hook

```
const DataFetcher = () => {
    const [data, setData] = useState(null);

    useEffect(() => {
        // Runs after component mounts
        fetchData().then(setData);

        // Cleanup function
        return () => {
            // Runs before unmount
            cleanup();
        };
    }, []); // Empty deps = run once

    return [
        "div",
        data ? ["p", data] : ["p", "Loading..."]
    ];
};
```

SuiWeb Implementation

SuiWeb Features

- Lightweight UI library
- Support for JSX and SJDON
- State and effect hooks
- Virtual DOM implementation
- Component lifecycle management

Component Patterns

```
// Container Component
const UserContainer = () => {
    const [user, setUser] = useState(null);

    useEffect(() => {
        fetchUser().then(setUser);
    }, []);

    return [UserProfile, {user}];
};

// Presentation Component
const UserProfile = ({user}) => {
    if (!user) return ["div", "Loading..."];

    return [
        "div",
        ["h2", user.name],
        ["p", user.email],
        [UserDetails, {details: user.details}]
    ];
};
```

Component Design Best Practices

Design Principles

- Single Responsibility Principle
- Separation of Concerns
- Container/Presentational Pattern
- Props Interface Design
- State Management Strategy

Component Architecture

```
// Bad: Mixed concerns
const UserCard = () => {
    const [user, setUser] = useState(null);

    useEffect(() => {
        fetchUser().then(setUser);
    }, []);

    return [
        "div", {className: "card"},
        ["h2", user?.name],
        ["p", user?.email]
    ];
};

// Good: Separated concerns
const UserCardContainer = () => {
    const [user, setUser] = useState(null);

    useEffect(() => {
        fetchUser().then(setUser);
    }, []);

    return [UserCardView, {user}];
};

const UserCardView = ({user}) => [
    "div", {className: "card"},
    ["h2", user?.name],
    ["p", user?.email]
];
```

Performance Optimization

- Virtual DOM diffing
- State updates batching
- Component memoization
- Lazy loading
- Event delegation

Advanced Patterns

```
// Higher Order Component
const withLoading = (WrappedComponent) => {
    return (props) => {
        const [loading, setLoading] = useState(true);

        useEffect(() => {
            setTimeout(() => setLoading(false), 1000);
        }, []);

        if (loading) return ["div", "Loading..."];
        return [WrappedComponent, props];
    };
};

// Compound Components
const Form = ({children}) => [
    "form",
    ...children
];

Form.Input = ({name, label}) => [
    "div",
    ["label", {for: name}, label],
    ["input", {id: name, name}]
];

Form.Submit = ({text}) => [
    "button",
    {type: "submit"},
    text
];
```

React Overview

React Features

- Component-based architecture
- Virtual DOM
- JSX syntax
- Hooks for state and effects
- Large ecosystem
- Active community

React vs SuiWeb

```
// React Component
const Counter = () => {
    const [count, setCount] = React.useState(0);

    return (
        <div>
            <h1>Count: {count}</h1>
            <button onClick={() => setCount(count +
                1)}>
                Increment
            </button>
        </div>
    );
};

// SuiWeb Component
const Counter = () => {
    const [count, setCount] = useState(0);

    return [
        "div",
        ["h1", `Count: ${count}`],
        ["button",
            {onclick: () => setCount(count + 1)},
            "Increment"
        ]
    ];
};
```

Course Wrap-up

Core Concepts Review

Key Technologies Primary building blocks of modern web development:

- **JavaScript**
  - Language fundamentals
  - Asynchronous programming
  - DOM manipulation
  - Modern features (ES6+)
- **Browser APIs**
  - DOM interface
  - Event handling
  - Web Storage
  - Fetch API
- **Component Architecture**
  - UI components
  - State management
  - Component lifecycle
  - Virtual DOM

Development Approaches Evolution of web development:

1. Static websites
2. Server-side rendering
3. Client-side JavaScript
4. Single Page Applications
5. Component-based frameworks

From SuiWeb to React

Framework Comparison **SuiWeb (Educational)**

- Simplified component model
- Basic state management
- SJDON notation
- Learning-focused implementation

React (Production)

- Complete framework ecosystem
- Advanced optimization features
- Large community and resources
- Production-ready tools

Migration Path

```
// SuiWeb Component
const Counter = () => {
    const [count, setCount] = useState(0);
    return [
        "div",
        ["h1", `Count: ${count}`],
        ["button",
            {onclick: () => setCount(count + 1)},
            "Increment"
        ]
    ];
};

// Equivalent React Component
const Counter = () => {
    const [count, setCount] = React.useState(0);
    return (
        <div>
            <h1>Count: {count}</h1>
            <button onClick={() => setCount(count +
                 1)}>
                Increment
            </button>
        </div>
    );
};
```

Advanced Topics

Further Areas Topics for continued learning:

- **Advanced JavaScript**
  - TypeScript
  - WebAssembly
  - Testing frameworks
- **Mobile Development**
  - Progressive Web Apps
  - React Native
  - Responsive design
- **Performance**
  - Code splitting
  - Lazy loading
  - Service workers

Best Practices Summary

Development Guidelines

- Write clean, maintainable code
- Follow component design principles
- Use modern JavaScript features
- Implement proper error handling
- Consider security implications
- Test code thoroughly
- Optimize performance
- Document code and APIs

Learning Resources

- Official Documentation
  - MDN Web Docs
  - React Documentation
  - Node.js Documentation
- Online Learning
  - freeCodeCamp
  - Frontend Masters
  - Eloquent JavaScript
- Community Resources
  - Stack Overflow
  - GitHub
  - Dev.to

Course Goals Achieved

- Understanding of JavaScript fundamentals
- Proficiency in DOM manipulation and browser APIs
- Component-based development skills
- Experience with modern web development tools
- Foundation for continued learning

Übungsaufgaben

JavaScript Grundlagen

Datentypen und Operatoren

Aufgabe 1: Was ist die Ausgabe folgender Ausdrücke?

```
typeof NaN
typeof []
typeof null
typeof undefined
[] == false
null === undefined
"5" + 3
"5" - 3
```

Lösung:

```
"number"     // NaN ist vom Typ number
"object"     // Arrays sind Objekte
"object"     // null ist historisch ein Objekt
"undefined"  // undefined ist ein eigener Typ
true         // [] wird zu 0 konvertiert
false        // === vergleicht auch Typen
"53"         // String-Konkatenation
2            // Numerische Subtraktion
```

Funktionen und Scoping

Aufgabe 2: Was ist die Ausgabe dieses Codes?

```
let x = 1;
const f = () => {
    let x = 2;
    return {
        getX: () => x,
        setX: (val) => { x = val; }
    };
};
const obj = f();
console.log(x);          // ?
console.log(obj.getX()); // ?
obj.setX(3);
console.log(obj.getX()); // ?
console.log(x);          // ?
```

Lösung:

```
1    // Globales x bleibt 1
2    // Closure hat Zugriff auf lokales x
3    // Lokales x wird auf 3 gesetzt
1    // Globales x bleibt unveraendert
```

DOM und Events

DOM Manipulation

Aufgabe 3: Erstellen Sie eine Funktion, die eine ToDo-Liste verwaltet.

```
function createTodoList(containerId) {
    // Container finden
    const container =
        document.getElementById(containerId);

    // Input und Liste erstellen
    const input = document.createElement('input');
    const button = document.createElement('button');
    const list = document.createElement('ul');

    // Button konfigurieren
    button.textContent = 'Add';
    button.onclick = () => {
        if (input.value.trim()) {
            const li = document.createElement('li');
            li.textContent = input.value;
            list.appendChild(li);
            input.value = '';
        }
    };

    // Elemente zusammenfuegen
    container.appendChild(input);
    container.appendChild(button);
    container.appendChild(list);
}
```

Event Handling

Aufgabe 4: Implementieren Sie einen Klick-Zähler mit Event Delegation.

```
document.getElementById('container').addEventListener('clic
    (e) => {
        if (e.target.matches('button')) {
            const count = (
                parseInt(e.target.dataset.count) || 0
            ) + 1;
            e.target.dataset.count = count;
            e.target.textContent = `Clicked ${count}
                times`;
        }
    }
);
```

Client-Server Kommunikation

Fetch API

Aufgabe 5: Implementieren Sie eine Funktion für API-Requests.

```
async function apiRequest(url, method = 'GET', data =
    null) {
    const options = {
        method,
        headers: {
            'Content-Type': 'application/json'
        }
    };

    if (data) {
        options.body = JSON.stringify(data);
    }

    try {
        const response = await fetch(url, options);
        if (!response.ok) {
            throw new Error(`HTTP error:
                ${response.status}`);
        }
        return await response.json();
    } catch (error) {
        console.error('API request failed:', error);
        throw error;
    }
}
```

Formular-Validierung

Aufgabe 6: Erstellen Sie eine Formular-Validierung.

```
function validateForm(formId) {
    const form = document.getElementById(formId);

    form.addEventListener('submit', (e) => {
        e.preventDefault();

        const formData = new FormData(form);
        const errors = [];

        // Email validieren
        const email = formData.get('email');
        if (!email.includes('@')) {
            errors.push('Invalid email');
        }

        // Passwort validieren
        const password = formData.get('password');
        if (password.length < 8) {
            errors.push('Password too short');
        }

        if (errors.length === 0) {
            // Form submission logic
            console.log('Form valid, submitting...');
            form.submit();
        } else {
            alert(errors.join('\n'));
        }
    });
}
```

UI-Komponenten

SuiWeb Komponente **Aufgabe 7:** Erstellen Sie eine Counter-Komponente mit SuiWeb.

```
const Counter = () => {
    const [count, setCount] = useState(0);

    return [
        "div",
        ["h2", `Count: ${count}`],
        ["button",
            {onclick: () => setCount(count + 1)},
            "Increment"
        ],
        ["button",
            {onclick: () => setCount(count - 1)},
            "Decrement"
        ]
    ];
};
```

Container Component **Aufgabe 8:** Implementieren Sie eine UserList-Komponente.

```
const UserList = () => {
    const [users, setUsers] = useState([]);
    const [loading, setLoading] = useState(true);

    if (loading) {
        fetchUsers()
            .then(data => {
                setUsers(data);
                setLoading(false);
            })
            .catch(error => {
                console.error(error);
                setLoading(false);
            });
    }

    if (loading) {
        return ["div", "Loading..."];
    }

    return [
        "div",
        ["h2", "Users"],
        ["ul",
            ...users.map(user =>
                ["li", `${user.name} (${user.email})`]
            )
        ]
    ];
};
```

Theoriefragen

Konzeptfragen **1. Erklären Sie den Unterschied zwischen == und === in JavaScript.**
Antwort: == vergleicht Werte mit Typumwandlung, === vergleicht Werte und Typen ohne Umwandlung.
2. Was ist Event Bubbling?
Antwort: Events werden von dem auslösenden Element durch den DOM-Baum nach oben weitergeleitet.
3. Was ist der Unterschied zwischen localStorage und sessionStorage?
Antwort: localStorage persistiert Daten auch nach Schließen des Browsers, sessionStorage nur während der Session.
4. Erklären Sie den Unterschied zwischen synchronem und asynchronem Code.
Antwort: Synchroner Code wird sequentiell ausgeführt, asynchroner Code ermöglicht parallele Ausführung ohne Blockierung.

Praktische Aufgaben

Implementierungsaufgaben **1. Implementieren Sie eine Funktion zur Deep Copy von Objekten.**
2. Erstellen Sie eine Funktion, die prüft ob ein String ein Palindrom ist.
3. Implementieren Sie eine debounce-Funktion.
4. Erstellen Sie eine Komponente für einen Image Slider.

Debugging-Aufgaben **1. Finden Sie den Fehler im folgenden Code:**

```
const getData = () => {
    fetch('api/data')
        .then(response => response.json())
        .then(data => {
            return data;
        });
}
// Warum kommt undefined zurueck?
```

Antwort: Die Funktion hat kein explizites return Statement. Sie sollte entweder async/await verwenden oder die Promise zurückgeben.

Example Exercises

JavaScript Fundamentals

Basic Array Manipulation Write a function that takes an array of numbers and returns a new array containing only the even numbers, doubled.

```
// Example solution
function processArray(numbers) {
    return numbers
        .filter(num => num % 2 === 0)
        .map(num => num * 2);
}

// Test
console.log(processArray([1, 2, 3, 4, 5, 6])); // [4,
    8, 12]
```

Closure Implementation Create a function that generates unique IDs with a given prefix. Each call should return a new ID with an incrementing number.

```
// Example solution
function createIdGenerator(prefix) {
    let counter = 0;
    return function() {
        counter++;
        return `${prefix}${counter}`;
    };
}

// Test
const generateUserId = createIdGenerator('user_');
console.log(generateUserId()); // "user_1"
console.log(generateUserId()); // "user_2"
```

Async Programming Write an async function that fetches user data from two different endpoints and combines them. Handle potential errors appropriately.

```
async function getUserData(userId) {
    try {
        const [profile, posts] = await Promise.all([
            fetch(`/api/profile/${userId}`).then(r =>
                r.json()),
            fetch(`/api/posts/${userId}`).then(r =>
                r.json())
        ]);

        return {
            ...profile,
            posts: posts
        };
    } catch (error) {
        console.error('Failed to fetch user data:',
            error);
        throw new Error('Failed to load user data');
    }
}
```

DOM Manipulation

Dynamic List Creation Write a function that takes an array of items and creates a numbered list in the DOM. Add a button to each item that removes it from the list.

```javascript
function createList(items, containerId) {
    const container =
        document.getElementById(containerId);
    const ul = document.createElement('ul');

    items.forEach((item, index) => {
        const li = document.createElement('li');
        li.textContent = `${index + 1}. ${item} `;

        const button =
            document.createElement('button');
        button.textContent = 'Remove';
        button.onclick = () => li.remove();

        li.appendChild(button);
        ul.appendChild(li);
    });

    container.appendChild(ul);
}
```

Component Implementation

Form Component Create a form component in SuiWeb that handles user input with validation and submits data to a server.

```javascript
const UserForm = () => {
    const [formData, setFormData] = useState({
        username: '',
        email: ''
    });
    const [errors, setErrors] = useState({});

    const validate = () => {
        const newErrors = {};
        if (!formData.username) {
            newErrors.username = 'Username is
                required';
        }
        if (!formData.email.includes('@')) {
            newErrors.email = 'Valid email is
                required';
        }
        setErrors(newErrors);
        return Object.keys(newErrors).length === 0;
    };

    const handleSubmit = async (e) => {
        e.preventDefault();
        if (!validate()) return;

        try {
            await fetch('/api/users', {
                method: 'POST',
                headers: {'Content-Type':
                    'application/json'},
                body: JSON.stringify(formData)
            });
        } catch (error) {
            setErrors({submit: 'Failed to submit
                form'});
        }
    };

    return [
        "form",
        {onsubmit: handleSubmit},
        ["div",
            ["label", {for: "username"}, "Username:"],
            ["input", {
                id: "username",
                value: formData.username,
                oninput: (e) => setFormData({
                    ...formData,
                    username: e.target.value
                })
            }],
            errors.username && ["span", {class:
                "error"}, errors.username]
        ],
        ["div",
            ["label", {for: "email"}, "Email:"],
            ["input", {
                id: "email",
                type: "email",
                value: formData.email,
                oninput: (e) => setFormData({
                    ...formData,
                    email: e.target.value
                })
            }],
            errors.email && ["span", {class: "error"},
                errors.email]
```

API Implementation

REST API with Express Create a simple REST API for a todo list with Express.js, including error handling and basic validation.

```javascript
const express = require('express');
const app = express();
app.use(express.json());

let todos = [];

// Get all todos
app.get('/api/todos', (req, res) => {
    res.json(todos);
});

// Create new todo
app.post('/api/todos', (req, res) => {
    const { title } = req.body;

    if (!title) {
        return res.status(400).json({
            error: 'Title is required'
        });
    }

    const todo = {
        id: Date.now(),
        title,
        completed: false
    };

    todos.push(todo);
    res.status(201).json(todo);
});

// Update todo
app.patch('/api/todos/:id', (req, res) => {
    const { id } = req.params;
    const { completed } = req.body;

    const todo = todos.find(t => t.id ===
        parseInt(id));

    if (!todo) {
        return res.status(404).json({
            error: 'Todo not found'
        });
    }

    todo.completed = completed;
    res.json(todo);
});

app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).json({
        error: 'Internal server error'
    });
});

app.listen(3000);
```

State Management Implement a shopping cart component that manages products, quantities, and total price calculation.

```javascript
const ShoppingCart = () => {
    const [items, setItems] = useState([]);

    const addItem = (product) => {
        setItems(current => {
            const existing = current.find(
                item => item.id === product.id
            );

            if (existing) {
                return current.map(item =>
                    item.id === product.id
                        ? {...item, quantity:
                            item.quantity + 1}
                        : item
                );
            }

            return [...current, {...product, quantity:
                1}];
        });
    };

    const removeItem = (productId) => {
        setItems(current =>
            current.filter(item => item.id !==
                productId)
        );
    };

    const total = items.reduce(
        (sum, item) => sum + item.price *
            item.quantity,
        0
    );

    return [
        "div",
        ["h2", "Shopping Cart"],
        ["ul",
            ...items.map(item => [
                "li",
                ["span", `${item.name} x
                    ${item.quantity}`],
                ["span", `$${item.price *
                    item.quantity}`],
                ["button",
                    {onclick: () =>
                        removeItem(item.id)},
                    "Remove"
                ]
            ])
        ],
        ["div", `Total: $${total.toFixed(2)}`]
    ];
};
```

Browser APIs and Events

Custom Event System Implement a publish/subscribe system using browser events.

```javascript
class EventBus {
    constructor() {
        this.eventTarget = new EventTarget();
    }

    publish(eventName, data) {
        const event = new CustomEvent(eventName, {
            detail: data,
            bubbles: true
        });
        this.eventTarget.dispatchEvent(event);
    }

    subscribe(eventName, callback) {
        const handler = (e) => callback(e.detail);
        this.eventTarget.addEventListener(eventName,
            handler);
        return () => {
            this.eventTarget.removeEventListener(eventName,
                handler);
        };
    }
}

// Usage
const bus = new EventBus();
const unsubscribe = bus.subscribe('userLoggedIn',
    (user) => {
    console.log(`Welcome, ${user.name}!`);
});

bus.publish('userLoggedIn', { name: 'John' });
unsubscribe(); // Cleanup
```

Drag and Drop Implement a simple drag and drop system for list items.

```javascript
function initDragAndDrop(containerId) {
    const container =
        document.getElementById(containerId);
    let draggedItem = null;

    container.addEventListener('dragstart', (e) => {
        draggedItem = e.target;
        e.target.classList.add('dragging');
    });

    container.addEventListener('dragend', (e) => {
        e.target.classList.remove('dragging');
    });

    container.addEventListener('dragover', (e) => {
        e.preventDefault();
        const afterElement =
            getDragAfterElement(container, e.clientY);
        if (afterElement) {
            container.insertBefore(draggedItem,
                afterElement);
        } else {
            container.appendChild(draggedItem);
        }
    });

    function getDragAfterElement(container, y) {
        const draggableElements = [
            ...container.querySelectorAll('li:not(.dragging)
        ];

        return draggableElements.reduce((closest,
            child) => {
            const box = child.getBoundingClientRect();
            const offset = y - box.top - box.height /
                2;

            if (offset < 0 && offset > closest.offset)
                {
                return { offset, element: child };
            }
            return closest;
        }, { offset: Number.NEGATIVE_INFINITY
            }).element;
    }
}
```

Data Manipulation and Algorithms

Deep Object Comparison Implement a function that deeply compares two objects for equality.

```javascript
function deepEqual(obj1, obj2) {
    // Handle primitives and null
    if (obj1 === obj2) return true;
    if (obj1 == null || obj2 == null) return false;
    if (typeof obj1 !== 'object' || typeof obj2 !==
        'object')
        return false;

    const keys1 = Object.keys(obj1);
    const keys2 = Object.keys(obj2);

    if (keys1.length !== keys2.length) return false;

    return keys1.every(key => {
        if (!keys2.includes(key)) return false;
        return deepEqual(obj1[key], obj2[key]);
    });
}

// Test
const obj1 = {
    a: 1,
    b: { c: 2, d: [3, 4] },
    e: null
};
const obj2 = {
    a: 1,
    b: { c: 2, d: [3, 4] },
    e: null
};
console.log(deepEqual(obj1, obj2)); // true
```

Custom Promise Implementation Create a simplified version of the Promise API.

```javascript
class MyPromise {
    constructor(executor) {
        this.state = 'pending';
        this.value = undefined;
        this.handlers = [];

        const resolve = (value) => {
            if (this.state === 'pending') {
                this.state = 'fulfilled';
                this.value = value;
                this.handlers.forEach(handler =>
                    this.handle(handler));
            }
        };

        const reject = (error) => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.value = error;
                this.handlers.forEach(handler =>
                    this.handle(handler));
            }
        };

        try {
            executor(resolve, reject);
        } catch (error) {
            reject(error);
        }
    }

    handle(handler) {
        if (this.state === 'pending') {
            this.handlers.push(handler);
        } else {
            const cb = this.state === 'fulfilled'
                ? handler.onSuccess
                : handler.onFail;
            if (cb) {
                try {
                    const result = cb(this.value);
                    handler.resolve(result);
                } catch (error) {
                    handler.reject(error);
                }
            }
        }
    }

    then(onSuccess, onFail) {
        return new MyPromise((resolve, reject) => {
            this.handle({
                onSuccess: onSuccess || (val => val),
                onFail: onFail || (err => { throw err;
                    }),
                resolve,
                reject
            });
        });
    }

    catch(onFail) {
        return this.then(null, onFail);
    }
}

// Usage
new MyPromise((resolve, reject) => {
    setTimeout(() => resolve('Success!'), 1000);
```

Component Testing

Unit Testing Components Write tests for a form component using Jasmine.

```javascript
describe('UserForm Component', () => {
    let form;

    beforeEach(() => {
        form = new UserForm();
    });

    it('should initialize with empty values', () => {
        expect(form.state.username).toBe('');
        expect(form.state.email).toBe('');
        expect(Object.keys(form.state.errors)).toHaveSize(0)
    });

    it('should validate email format', () => {
        form.state.email = 'invalid-email';
        const isValid = form.validate();

        expect(isValid).toBe(false);
        expect(form.state.errors.email)
            .toContain('Valid email is required');
    });

    it('should submit form with valid data', async ()
        => {
        form.state.username = 'testuser';
        form.state.email = 'test@example.com';

        spyOn(window, 'fetch').and.returnValue(
            Promise.resolve({ ok: true })
        );

        await form.handleSubmit();

        expect(window.fetch).toHaveBeenCalledWith(
            '/api/users',
            jasmine.any(Object)
        );
        expect(form.state.errors).toEqual({});
    });
});
```