Implementing Word-like features in web-based rich text editors

Building a web editor with Microsoft Word-like functionality requires implementing **toolbar state synchronization**, **indentation controls**, **text formatting**, and **document theming**. Tiptap built on ProseMirror offers the best balance of extensibility and development speed for this purpose, with extensive community extensions and a proven architecture. This guide provides complete code patterns, library recommendations, and build-versus-adopt analysis for each feature area.

Toolbar state synchronization with selection

Web editors must update toolbar button states (bold active, current heading level, alignment) whenever the cursor moves or selection changes. This differs fundamentally from desktop Word because web editors use event-driven reactive patterns rather than direct DOM inspection.

Tiptap's `editor.isActive()` pattern

Tiptap provides a declarative API for checking formatting state:

```typescript
// Basic mark checking
editor.isActive('bold')
editor.isActive('italic')
editor.isActive('heading', { level: 1 })
editor.isActive('textAlign', { textAlign: 'center' })

// Getting current attributes
const fontSize = editor.getAttributes('textStyle').fontSize
const fontFamily = editor.getAttributes('textStyle').fontFamily
```

For React, the `useEditorState` hook provides efficient reactive updates: Tiptap Tiptap

```typescript
import { useEditor, useEditorState } from '@tiptap/react'

function Toolbar({ editor }) {
  const state = useEditorState({
    editor,
    selector: ({ editor }) => ({
      isBold: editor.isActive('bold'),
      isItalic: editor.isActive('italic'),
      headingLevel: [1,2,3,4,5,6].find(l =>
        editor.isActive('heading', { level: l })
      ) || null,
      textAlign: ['left','center','right','justify'].find(a =>
        editor.isActive({ textAlign: a })
      ) || 'left',
      isBulletList: editor.isActive('bulletList'),
      isOrderedList: editor.isActive('orderedList'),
      fontFamily: editor.getAttributes('textStyle').fontFamily,
      fontSize: editor.getAttributes('textStyle').fontSize,
    }),
  })

  return (
    <button
      className={state.isBold ? 'active' : ''}
      onClick={() => editor.chain().focus().toggleBold().run()}
    >
      Bold
    </button>
  )
}
```

Performance optimization: For documents over **3,000 lines**, `isActive()` calls can take 14-110ms each. GitHub Batch all state checks in a single selector function and consider debouncing updates by 50ms on very large documents.

ProseMirror lower-level approach

For direct ProseMirror implementations, check marks against the current selection:

```typescript
```

```
function isMarkActive(state: EditorState, markType: MarkType): boolean {
  const { from, $from, to, empty } = state.selection

  if (empty) {
    // Check stored marks first, then position marks
    if (state.storedMarks) {
      return markType.isInSet(state.storedMarks) !== null
    }
    return markType.isInSet($from.marks()) !== null
  }
  return state.doc.rangeHasMark(from, to, markType)
}
```

Cross-editor comparison

Editor	State API	Update Mechanism
Tiptap	`editor.isActive()`	`useEditorState` hook
ProseMirror	Custom `isMarkActive()`	Plugin view `update()`
Slate.js	`Editor.marks()`	`useSlate()` hook
CKEditor 5	`command.value` observable	Automatic binding
TinyMCE	`formatter.match()`	`formatChanged()` callback

Reliable indent/outdent and paragraph spacing

Microsoft Word uses a sophisticated indentation model with first-line indent, hanging indent, and left/right paragraph margins. Web implementations typically simplify this to discrete indent levels.

Complete Tiptap indent extension

```typescript
```

```
import { Extension } from '@tiptap/core'
import { TextSelection, AllSelection, Transaction } from 'prosemirror-state'

export const Indent = Extension.create({
  name: 'indent',

  addOptions() {
    return {
      types: ['heading', 'paragraph'],
      minIndent: 0,
      maxIndent: 7,
    }
  },

  addGlobalAttributes() {
    return [{
      types: this.options.types,
      attributes: {
        indent: {
          default: 0,
          renderHTML: attrs => {
            if (!attrs.indent) return {}
            return {
              'data-indent': attrs.indent,
              style: `margin-left: ${attrs.indent * 2}em`
            }
          },
          parseHTML: el => ({
            indent: parseInt(el.getAttribute('data-indent') || '0')
          })
        }
      }
    }]
  },

  addCommands() {
    return {
      indent: () => ({ tr, state, dispatch }) => {
        const { from, to } = state.selection
        state.doc.nodesBetween(from, to, (node, pos) => {
          if (this.options.types.includes(node.type.name)) {
            const newIndent = Math.min(
              (node.attrs.indent || 0) + 1,
              this.options.maxIndent
            )
            tr = tr.setNodeMarkup(pos, null, {
              ...node.attrs,
              indent: newIndent
            })
          }
        })
        if (dispatch) dispatch(tr)
        return true
      },
      outdent: () => ({ tr, state, dispatch }) => {
        const { from, to } = state.selection
        state.doc.nodesBetween(from, to, (node, pos) => {
          if (this.options.types.includes(node.type.name)) {
            const newIndent = Math.max(
              (node.attrs.indent || 0) - 1,
              this.options.minIndent
            )
            tr = tr.setNodeMarkup(pos, null, {
              ...node.attrs,
              indent: newIndent
            })
          }
        })
        if (dispatch) dispatch(tr)
        return true
      }
    }
  },

  addKeyboardShortcuts() {
    return {
      'Tab': () => this.editor.commands.indent(),
      'Shift-Tab': () => this.editor.commands.outdent(),
    }
  }
})
```

Paragraph spacing extension

```typescript
export const ParagraphSpacing = Extension.create({
  name: 'paragraphSpacing',

  addGlobalAttributes() {
    return [{
      types: ['paragraph', 'heading'],
      attributes: {
        spacingBefore: {
          default: 0,
          renderHTML: attrs => ({ style: `margin-top: ${attrs.spacingBefore}pt` }),
        },
        spacingAfter: {
          default: 10,
          renderHTML: attrs => ({ style: `margin-bottom: ${attrs.spacingAfter}pt` }),
        },
        lineHeight: {
          default: '1.5',
          renderHTML: attrs => ({ style: `line-height: ${attrs.lineHeight}` }),
        }
      }
    }]
  },

  addCommands() {
    return {
      setLineHeight: (value: string) => ({ commands }) =>
        commands.updateAttributes('paragraph', { lineHeight: value }),
      setSpacingBefore: (value: number) => ({ commands }) =>
        commands.updateAttributes('paragraph', { spacingBefore: value }),
      setSpacingAfter: (value: number) => ({ commands }) =>
        commands.updateAttributes('paragraph', { spacingAfter: value }),
    }
  }
})
```

Enter vs Shift+Enter handling

Tiptap's `HardBreak` extension handles Shift+Enter for soft line breaks (`<br>`), `Tiptap` while Enter creates new paragraphs. Configure with:

```typescript
import HardBreak from '@tiptap/extension-hard-break'

HardBreak.configure({
  keepMarks: true,  // Preserve formatting across break
})

// Default: Shift+Enter = hard break, Enter = new paragraph
// To swap behavior:
const CustomEnter = Extension.create({
  addKeyboardShortcuts() {
    return {
      'Enter': () => this.editor.commands.setHardBreak(),
      'Shift-Enter': () => this.editor.commands.splitBlock(),
    }
  }
})
```

Case transforms and font size implementation

Microsoft Word's Shift+F3 cycles through: **lowercase → UPPERCASE → Title Case → lowercase**. The Change Case dialog adds Sentence case and tOGGLE cASE. For web editors, use JavaScript character transformation rather than CSS `text-transform` because CSS only changes visual presentation—copy/paste preserves original text.

Complete case transformation utilities

```javascript
```

```
const caseTransforms = {
  toLowerCase: (str) => str.toLowerCase(),

  toUpperCase: (str) => str.toUpperCase(),

  toSentenceCase: (str) =>
    str.toLowerCase().replace(/(^\s*\w|[.!?]\s*\w)/g, m => m.toUpperCase()),

  toTitleCase: (str) =>
    str.toLowerCase().replace(/\b\w/g, c => c.toUpperCase()),

  toToggleCase: (str) =>
    str.split('').map(c =>
      c === c.toUpperCase() ? c.toLowerCase() : c.toUpperCase()
    ).join('')
}

// Word-style Shift+F3 cycle
function cycleCase(str, current = 'original') {
  const cycle = {
    'original': 'lowercase',
    'lowercase': 'uppercase',
    'uppercase': 'titlecase',
    'titlecase': 'lowercase'
  }
  const next = cycle[current] || 'lowercase'
  return {
    text: caseTransforms[`to${next.charAt(0).toUpperCase() + next.slice(1)}`](str),
    nextState: next
  }
}
```

Applying case transforms in Tiptap

```typescript
function applyCaseTransform(editor, transform) {
  const { from, to } = editor.state.selection
  const selectedText = editor.state.doc.textBetween(from, to)
  const transformed = transform(selectedText)

  editor.chain()
    .focus()
    .deleteRange({ from, to })
    .insertContent(transformed)
    .run()
}

// Usage
applyCaseTransform(editor, caseTransforms.toUpperCase)
```

Font size mark implementation

```typescript
```

```
import { Extension } from '@tiptap/core'
import '@tiptap/extension-text-style'

export const FontSize = Extension.create({
 name: 'fontSize',

 addOptions() {
  return { types: ['textStyle'] }
 },

 addGlobalAttributes() {
  return [{
   types: this.options.types,
   attributes: {
    fontSize: {
     default: null,
     parseHTML: el => el.style.fontSize?.replace(/['"]+/g, ''),
     renderHTML: attrs =>
      attrs.fontSize ? { style: `font-size: ${attrs.fontSize}` } : {}
    }
   }
  }]
 },

 addCommands() {
  return {
   setFontSize: (size) => ({ chain }) =>
    chain().setMark('textStyle', { fontSize: size }).run(),
   unsetFontSize: () => ({ chain }) =>
    chain().setMark('textStyle', { fontSize: null }).removeEmptyTextStyle().run()
  }
 }
})
```

Small caps via CSS mark

```typescript
export const SmallCaps = Mark.create({
 name: 'smallCaps',

 parseHTML() {
  return [{
   tag: 'span',
   getAttrs: el => el.style.fontVariant === 'small-caps' ? {} : false
  }]
 },

 renderHTML() {
  return ['span', { style: 'font-variant-caps: small-caps' }, 0]
 },

 addCommands() {
  return {
   toggleSmallCaps: () => ({ commands }) => commands.toggleMark(this.name)
  }
 }
})
```

Microsoft Word OOXML theme structure

Word themes are stored in `.docx` files at `word/theme/theme1.xml`. File-extensions The theme defines **12 color slots** and **2 font families** that cascade through all document styles. Officeopenxml

Theme color slots

Slot	Purpose	Default (Office 2023)
dk1	Primary text	#000000 (system)
lt1	Primary background	#FFFFFF (system)
dk2	Secondary dark	#44546A
lt2	Secondary light	#E7E6E6
accent1	Headings, emphasis	#4472C4
accent2-6	Charts, SmartArt	Various
hlink	Hyperlinks	#0563C1

Slot	Purpose	Default (Office 2023)
folHlink	Visited links	#954F72

Font scheme structure

Themes define **majorFont** for headings and **minorFont** for body text:

```xml
<a:fontScheme name="Office">
  <a:majorFont>
    <a:latin typeface="Aptos Display"/>
  </a:majorFont>
  <a:minorFont>
    <a:latin typeface="Aptos"/>
  </a:minorFont>
</a:fontScheme>
```

Tint/shade color modifications

Word applies themeTint (lighter) and themeShade (darker) modifiers to base colors:

```javascript
// themeTint: value 0-255, makes color lighter
function applyTint(rgb, tintValue) {
  const tint = tintValue / 255
  return {
    r: Math.round(rgb.r + (255 - rgb.r) * tint),
    g: Math.round(rgb.g + (255 - rgb.g) * tint),
    b: Math.round(rgb.b + (255 - rgb.b) * tint)
  }
}

// themeShade: value 0-255, makes color darker (applied to luminance)
function applyShade(hsl, shadeValue) {
  const shade = shadeValue / 255
  return { ...hsl, l: hsl.l * shade }
}
```

Implementing themes in web editors with CSS variables

```css
:root {
  /* Theme colors */
  --theme-dk1: #000000;
  --theme-lt1: #FFFFFF;
  --theme-accent1: #4472C4;
  --theme-accent1-shade-75: #2F5496;
  --theme-hlink: #0563C1;

  /* Theme fonts */
  --theme-major-font: 'Aptos Display', sans-serif;
  --theme-minor-font: 'Aptos', sans-serif;
}

h1, h2, h3 {
  font-family: var(--theme-major-font);
  color: var(--theme-accent1-shade-75);
}

p { font-family: var(--theme-minor-font); }
a { color: var(--theme-hlink); }
a:visited { color: var(--theme-folHlink); }
```

Open-source editor framework comparison

After evaluating 8 major frameworks, **Tiptap** emerges as the best choice for most Word-like implementations due to its balance of extensibility, documentation, and ecosystem.

Feature comparison matrix

Feature	ProseMirror	Tiptap	Slate.js	CKEditor 5	Quill	Lexical
License	MIT	MIT	MIT	GPL/Commercial	BSD-3	MIT
GitHub Stars	~7.6k	~33.7k	~31.3k	~10k	~46.5k	~22.5k

Feature	ProseMirror	Tiptap	Slate.js	CKEditor 5	Quill	Lexical
npm Downloads/wk	~350k	~1.5M	~1.6M	~500k	~2.7M	~1.6M
Schema System	Robust	Via extensions	JSON-based	Plugin-based	Blots	Node-based
Collaboration	Excellent (Yjs)	Excellent	Good	Proprietary	Good	Good
Bundle Size	~50KB	~20KB core	~40KB	~150KB+	~70KB	~50KB
Learning Curve	Steep	Moderate	Moderate	Moderate	Easy	Moderate

When to use each framework

- **Tiptap**: Best for most projects—active ecosystem, excellent DX, extensible liveblocks
- **ProseMirror**: Maximum control for complex requirements, best collaboration support
- **CKEditor 5**: Enterprise projects with budget for commercial license, full feature set
- **Slate.js** (with Plate): React-only projects needing deep UI customization
- **Quill**: Quick prototypes, simple editing needs
- **Lexical**: New React projects (Meta backing, but pre-1.0)
- **Draft.js**: ❌ Archived—migrate to Lexical

Tiptap vs raw ProseMirror trade-offs

Aspect	Tiptap	Raw ProseMirror
Setup time	Minutes	Hours/days
Extensions ecosystem	100+ available	Build from scratch
Abstraction cost	Adds layers, may hide issues	Direct control
Advanced features	May need ProseMirror knowledge	Full access

Recommendation: Start with Tiptap, drop to ProseMirror level only when needed.

Essential npm packages and UI kits

Tiptap official extensions for Word-like features

```bash
npm install @tiptap/react @tiptap/starter-kit \
  @tiptap/extension-text-style @tiptap/extension-color \
  @tiptap/extension-font-family @tiptap/extension-text-align \
  @tiptap/extension-underline @tiptap/extension-highlight \
  @tiptap/extension-table @tiptap/extension-table-row \
  @tiptap/extension-table-cell @tiptap/extension-table-header
```

Pre-built UI kits

Package	Framework	Features
mui-tiptap	Material UI	Complete toolbar, ColorPicker, TableImproved
@mantine/tiptap	Mantine	Full toolbar, Link controls
vuetify-pro-tiptap	Vuetify	Vue3 integration

Document import/export

Package	Purpose	Downloads/wk
docx	Create .docx programmatically GitHub	~693K
mammoth	Convert .docx to HTML	High
docxtemplater	Template-based generation npm	High

Color/font pickers

Package	Description
react-colorful	Lightweight (2KB), 12+ color models (npm)
react-color	13 picker types (Sketch, Chrome, etc.) (npm)
font-picker-react	Google Fonts integration (npm)

Community Tiptap extensions worth noting

- **tiptap-extension-office-paste**: Office paste handling
- **tiptap-extension-pagination**: Page breaks
- **@gocapsule/column-extension**: Multi-column layouts
- **tiptap-footnotes**: Footnotes support

Complete integration example

typescript

```typescript
import { useEditor, EditorContent } from '@tiptap/react'
import StarterKit from '@tiptap/starter-kit'
import TextStyle from '@tiptap/extension-text-style'
import Color from '@tiptap/extension-color'
import FontFamily from '@tiptap/extension-font-family'
import TextAlign from '@tiptap/extension-text-align'
import Underline from '@tiptap/extension-underline'
import Table from '@tiptap/extension-table'
import TableRow from '@tiptap/extension-table-row'
import TableCell from '@tiptap/extension-table-cell'
import { Indent } from './extensions/Indent'
import { FontSize } from './extensions/FontSize'
import { ParagraphSpacing } from './extensions/ParagraphSpacing'

const WordLikeEditor = () => {
  const editor = useEditor({
    extensions: [
      StarterKit,
      Underline,
      TextStyle,
      Color,
      FontFamily.configure({ types: ['textStyle'] }),
      FontSize,
      TextAlign.configure({ types: ['heading', 'paragraph'] }),
      Table.configure({ resizable: true }),
      TableRow,
      TableCell,
      Indent,
      ParagraphSpacing,
    ],
    content: '<p>Start editing...</p>',
  })

  return (
    <div className="editor-container">
      <Toolbar editor={editor} />
      <EditorContent editor={editor} />
    </div>
  )
}
```

Build vs adopt recommendations

Decision matrix

Scenario	Recommendation	Time to MVP
Enterprise document management	Adopt OnlyOffice/Collabora	2-4 weeks
Notion-like note app	Build with Tiptap + Novel patterns	1-2 months
Blog/CMS editor	Tiptap + StarterKit + mammoth	2-4 weeks
Email composer	mui-tiptap hybrid	2-3 weeks
Full Word clone	OnlyOffice (OSS)	Immediate

Scenario	Recommendation	Time to MVP
SaaS with custom branding	Tiptap hybrid	1-3 months

Hybrid approach: recommended stack

For most projects needing Word-like features with custom branding:

```javascript
// Core editor
import { useEditor } from '@tiptap/react'
import StarterKit from '@tiptap/starter-kit'

// Formatting
import TextStyle from '@tiptap/extension-text-style'
import Color from '@tiptap/extension-color'
import FontFamily from '@tiptap/extension-font-family'

// UI kit
import { RichTextEditor } from 'mui-tiptap' // or @mantine/tiptap

// Import/export
import mammoth from 'mammoth'  // .docx → HTML
import { Document, Packer } from 'docx'  // HTML → .docx
```

Pros of hybrid: Fast start with pre-built UI (1-3 months), full customization where needed, selective Pro feature adoption.

Cons: Multiple dependencies to manage, some integration work required.

Cost considerations

- **Tiptap Pro** (collaboration, export): ~$299-999/month
- **CKEditor Commercial**: Based on editor loads
- **OnlyOffice Enterprise**: ~$1500+/year
- **Full custom build**: 3-6 months dev time

Conclusion

Implementing Word-like features in web editors is achievable with modern frameworks, though no single solution replicates Word completely out-of-box. **Tiptap provides the strongest foundation** for custom implementations— `Liveblocks`its extension architecture maps well to Word's feature set, and the community has solved many common challenges.

For toolbar state, use `useEditorState` with batched `isActive()` calls. For indentation, implement a custom extension with `data-indent` attributes and margin-based styling. For themes, map Word's 12-color scheme to CSS variables and apply through style attributes. For document export, combine `docx` (creation) and `mammoth` (import) packages.

The hybrid approach—starting with Tiptap plus a UI kit like mui-tiptap, adding custom extensions for specific features—delivers the best balance of development speed and customization for most use cases.