2. Setup Process

2.1 Initial Setup

I began by creating the basic project structure with:

- index.html for the UI
- index.js containing the ES6 class-based application logic
- Configuration files for Webpack and Babel

2.2 Tool Configuration

The key configuration steps included:

Babel Setup:

- Installed @babel/preset-env with specific browser targets
- Configured core-js for polyfills with useBuiltIns: "usage" to automatically include required polyfills

Webpack Configuration:

- Set up basic bundling with Babel loader
- Defined entry point and output bundle location
- Added production and development build scripts

3. Challenges and Solutions

3.1 Challenge: Internet Explorer 11 Support

Problem: The initial build failed to work in IE11, throwing syntax errors for arrow functions and class declarations.

Solution:

- Added explicit browser targets in Babel config ("ie >= 11")
- Ensured core-js was properly installed and configured
- Verified polyfills were being included in the final bundle

3.2 Challenge: Class Properties

Problem: The class methods weren't being transpiled correctly, causing errors in older browsers.

Solution:

- Added @babel/plugin-proposal-class-properties (though ultimately removed as it wasn't needed for basic class syntax)
- Simplified class method definitions to ensure compatibility

3.3 Challenge: Live Reloading

Problem: During development, changes weren't automatically reflected.

Solution:

- Added webpack --watch mode through the start script
- Configured proper file watching and cache busting

4. Build Process Optimization

4.1 Production vs Development

- Created separate build modes:
  - Production: Minified and optimized
  - Development: With source maps for debugging

4.2 Bundle Analysis

- Initially the bundle size was larger than expected due to including all possible polyfills
- Optimized by using useBuiltIns: "usage" to only include necessary polyfills

5. Testing and Validation

5.1 Browser Testing

The application was tested in:

- Modern browsers (Chrome, Firefox, Edge)
- Legacy browsers (Internet Explorer 11)
- Mobile browsers (Safari iOS, Chrome Android)

5.2 Functionality Verification

All core features were verified:

- Adding new todos

- Marking todos as complete
- Deleting todos
- Persistent state (during page refresh)

6. Lessons Learned

1. **Polyfill Management:** Proper configuration of core-js is crucial for cross-browser support. The usage option significantly reduces bundle size.
2. **Build Tooling:** Webpack and Babel require careful configuration to work together effectively. The order of loaders and presets matters.
3. **Legacy Browser Support:** Testing in IE11 early in the process helps identify issues before they become deeply embedded in the codebase.
4. **Development Workflow:** Setting up proper watch modes and source maps from the beginning saves significant debugging time.

7. Conclusion

The project successfully demonstrates how modern JavaScript can be used while maintaining compatibility with older browsers. The combination of ES6 classes, Webpack for bundling, and Babel for transpilation creates a robust foundation for cross-browser web applications. The challenges encountered were primarily related to configuration rather than application logic, highlighting the importance of proper build tool setup in modern web development.