**How the Auto-Save Feature Works**

The auto-save functionality I've implemented works in several layers to ensure your work is preserved without interrupting your creative flow. Here's a detailed explanation:

**1. Auto-Save Initialization**

When the application starts:

* The auto-save system initializes in the mounted() lifecycle hook
* It checks for any previously auto-saved data and offers to restore it
* It sets up a timer to periodically save your work

**2. Save Triggers**

The auto-save is triggered in two ways:

**Periodic Saving:**

* A timer runs every 60 seconds (configurable via autoSaveInterval)
* This ensures regular backups even during periods of inactivity

**Change-Based Saving:**

* Every time you make a change to the canvas (add, edit, or delete shapes)
* The addHistory() method marks the document as modified
* To prevent excessive saves during rapid editing, there's a 5-second debounce timer
* This means it waits until you've stopped editing for 5 seconds before saving

**3. Save Process**

When a save is triggered:

1. The status indicator shows "Saving..." with a spinner animation
2. The current document state is captured, including:
   * All shapes with their properties
   * Layer information
   * Canvas settings (grid size, rulers, etc.)
3. This data is serialized to JSON
4. The data is stored in the browser's localStorage with a key based on the document name
5. When complete, the status changes to "Saved at [time]" with a checkmark
6. The indicator automatically hides after 3 seconds

**4. Error Handling**

If a save fails:

* The status indicator shows "Save failed!" with an error icon
* A notification appears with details about the error
* You're given options to retry the save or save manually
* The system will continue attempting periodic saves

**5. Data Recovery**

When you reopen the application:

1. It checks localStorage for any saved data
2. If found, it shows a notification with when the data was saved
3. You can choose to restore the saved work or ignore it
4. If restored, all your shapes, layers, and settings are recovered exactly as you left them

**6. Visual Feedback**

The system provides clear visual feedback:

* A status indicator in the bottom status bar shows the current save state
* Notifications appear for important events (recovery options, errors)
* The indicator uses color coding (blue for saving, green for saved, red for errors)

This comprehensive auto-save system ensures your work is protected against browser crashes, accidental closures, or power failures, all while staying out of your way during the creative process.

ere's a summary of what we've accomplished:

1. **Added image placeholder functionality** - Created a helper method to display visual placeholders for images in different states (loading, error, empty)
2. **Created notification components**:
   * Built a reusable Notification component for displaying user-friendly messages
   * Implemented a NotificationManager to handle multiple notifications
3. **Implemented auto-save functionality**:
   * Added periodic auto-saving (every 60 seconds)
   * Added change-based auto-saving with debounce
   * Integrated auto-save status directly into the existing status bar
   * Added visual indicators for different states (saving, saved, error)
4. **Added data recovery**:
   * Implemented localStorage-based persistence
   * Added recovery options when auto-saved data is found

The auto-save status now appears in the existing status bar with appropriate styling and icons for each state:

* Blue indicator when saving
* Green checkmark when saved successfully
* Red X when there's an error

This implementation maintains the clean UI of your application while adding important functionality to prevent data loss.

**1. Canvas Caching for Static Elements**

We've implemented a static canvas that caches the background and grid, which are only redrawn when necessary:

* Created a separate offscreen canvas (staticCanvas) to store static elements
* Added tracking variables to detect when the static canvas needs updating
* Implemented a \_renderStaticElements() method that draws the background and grid to the static canvas
* Modified the main render method to use the cached static canvas instead of redrawing everything

This significantly improves performance because the background and grid (which can be complex to draw, especially with many grid lines) are only redrawn when zoom, offset, or grid properties change.

**2. RequestAnimationFrame for Smoother Rendering**

We've optimized the animation and rendering loop:

* Used requestAnimationFrame for all animations to sync with the browser's refresh rate
* Added proper cancellation of animation frames when the component is destroyed
* Tracked animation frame IDs to prevent memory leaks

**3. Optimized Render Loop**

We've made the render loop more efficient:

* Added a flag system to only trigger renders when necessary
* Implemented dirty checking to avoid unnecessary redraws
* Separated static and dynamic elements to minimize redrawing
* Added performance monitoring to detect slow renders

**Additional Improvements**

* Added proper cleanup in the beforeUnmount lifecycle hook
* Improved the animation method to be more efficient
* Added watchers to detect when the static canvas needs updating
* Optimized the main render method to avoid unnecessary work

These changes should result in:

1. **Smoother panning and zooming** - Especially noticeable during continuous operations
2. **Reduced CPU usage** - By avoiding unnecessary redraws
3. **Better performance with complex drawings** - The static elements are cached
4. **More responsive UI** - The main thread is less burdened

The application now starts with a clean canvas and has significantly improved rendering performance, especially for operations like dragging, zooming, and working with many shapes.

More added features.

**1. Shape Caching for Complex Objects**

For complex shapes (especially paths with many points or text with special formatting), consider implementing shape-level caching:

**2. Implement Viewport Culling**

Only render shapes that are visible in the current viewport:

**3. Implement Web Workers for Heavy Operations**

**4. Add Debounced Event Handlers**

**5. Implement Progressive Loading for Large Documents**

**6. Add Memory Management**

**7. Add Performance Monitoring**

These enhancements will further improve performance, especially for complex documents with many shapes or when working with large images. They build on the optimizations we've already implemented and provide a solid foundation for scaling your application.

1. **Shape Caching for Complex Objects**
   * Added a caching system for complex shapes like text, images, and shapes with rotation
   * Implemented methods to determine which shapes should be cached
   * Added timestamp-based cache expiration to prevent memory leaks
2. **Viewport Culling**
   * Added a method to determine if shapes are visible in the current viewport
   * Skip rendering shapes that are outside the visible area
   * Added padding to prevent popping at viewport edges
3. **Performance Monitoring**
   * Added detailed logging of render statistics
   * Track number of shapes rendered, skipped, and cached
   * Only log when render time exceeds a threshold
4. **Optimized Rendering Loop**
   * Used requestAnimationFrame for smoother animations
   * Added proper cleanup of animation frames
   * Implemented dirty checking to avoid unnecessary redraws

These optimizations should significantly improve the performance of the application, especially when working with complex drawings containing many shapes.

1. **Performance Metrics Tracking**
   * Added detailed metrics tracking for render times, shape counts, and cache usage
   * Implemented running averages and peak detection
   * Added counters for shapes rendered, skipped, and cached
2. **Debug Overlay**
   * Created a visual overlay that displays all performance metrics
   * Designed with a semi-transparent background and clear formatting
   * Shows real-time data about the canvas state and rendering performance
3. **Enhanced Debug Button**
   * Updated the debug button to toggle the overlay
   * Added visual feedback when debug mode is active
   * Button text changes to indicate current state
4. **Comprehensive Console Logging**
   * Added detailed logging of all canvas state
   * Included cache statistics and performance metrics
   * Organized information into logical categories

These changes provide a powerful debugging tool that will help you:

1. Identify performance bottlenecks
2. Monitor the effectiveness of the caching system
3. See how many shapes are being culled by the viewport system
4. Track render times to ensure smooth performance

To use it:

1. Click the "Debug" button in the status bar
2. The overlay will appear showing all performance metrics
3. Click "Hide Debug" to turn it off

This makes it much easier to monitor performance in real-time without having to open the browser console.

npx playwright test --project=chromium