<tldraw/>

Editor

The `Editor` class is the main way of controlling tldraw's editor. You can use it to manage the editor's internal state, make changes to the document, or respond to changes that have occurred.

By design, the `Editor`'s surface area is very large. Almost everything is available through it. Need to create some shapes? Use `Editor.createShapes`. Need to delete them? Use `Editor.deleteShapes`. Need a sorted array of every shape on the current page? Use `Editor.getCurrentPageShapesSorted`.

This page gives a broad idea of how the `Editor` class is organized and some of the architectural concepts involved. The full reference is available in the `Editor` API.

Store

The editor holds the raw state of the document in its `Editor.store` property. Data is kept here as a table of JSON serializable records.

For example, the store contains a `TLPage` record for each page in the current document, as well as an `TLInstancePageState` record for each page that stores information about the editor's state for that page, and a single `TLInstance` for each editor instance which stores the id of the user's current page.

The editor also exposes many *computed* values which are derived from other records in the store. For example, `Editor.getSelectedShapeIds` is a method that returns the editor's current selected shape ids for its current page.

You can use these properties directly or you can use them in signals.

```
import { track, useEditor } from 'tldraw'
```

```
export const SelectedShapeIdsCount = track(() => {
  const editor = useEditor()

  return <div>{editor.getSelectedShapeIds().length}</div>
})
```

Changing the state

The `Editor` class has many methods for updating its state. For example, you can change the current page's selection using `Editor.setSelectedShapes`. You can also use other convenience methods, such as `Editor.select`, `Editor.selectAll`, or `Editor.selectNone`.

```
editor.selectNone()
editor.select(myShapeId, myOtherShapeId)
editor.getSelectedShapeIds() // [myShapeId, myOtherShapeId]
```

Each change to the state happens within a transaction. You can batch changes into a single transaction using the `Editor.batch` method. It's a good idea to batch wherever possible, as this reduces the overhead for persisting or distributing those changes.

Listening for changes

You can subscribe to changes using the `Store.listen` method on `Editor.store`. Each time a transaction completes, the editor will call the callback with a history entry. This entry contains information about the records that were added, changed, or deleted, as well as whether the change was caused by the user or from a remote change.

```
editor.store.listen((entry) => {
  entry // { changes, source }
})
```

Remote changes

By default, changes to the editor's store are assumed to have come from the editor itself. You can use the `Store.mergeRemoteChanges` method of the editor's `Editor.store` to make changes in the store that will be emitted via `Store.listen` with the `source` property as `'remote'`.

If you're setting up some kind of multiplayer backend, you would want to send only the `'user'` changes to the server and merge the changes from the server using `Store.mergeRemoteChanges` (`editor.store.mergeRemoteChanges`).

Undo and redo

The history stack in tldraw contains two types of data: "marks" and "commands". Commands have their own `undo` and `redo` methods that describe how the state should change when the command is undone or redone.

You can call `Editor.mark` to add a mark to the history stack with the given `id`.

```
editor.mark('my-id')
// do some stuff
editor.bailToMark('my-id')
```

When you call `Editor.undo`, the editor will undo each command until it finds either a mark or the start of the stack. When you call `Editor.redo`, the editor will redo each command until it finds either a mark or the end of the stack.

```
// A
editor.mark('duplicate everything')
editor.selectAll()
editor.duplicateShapes(editor.getSelectedShapeIds())
// B
```

```
editor.undo() // will return to A
editor.redo() // will return to B
```

You can call `Editor.bail` to undo and delete all commands in the stack until the first mark.

```
// A
editor.mark('duplicate everything')
editor.selectAll()
editor.duplicateShapes(editor.getSelectedShapeIds())
// B

editor.bail() // will return to A
editor.redo() // will do nothing
```

You can use `Editor.bailToMark` to undo and delete all commands and marks until you reach a mark with the given `id`.

```
// A
editor.mark('first')
editor.selectAll()
// B
editor.mark('second')
editor.duplicateShapes(editor.getSelectedShapeIds())
// C

editor.bailToMark('first') // will return to A
```

Events

The `Editor` class receives events from its `Editor.dispatch` method. When the `Editor` receives an event, it is first handled internally to update `Editor.inputs` and other state before, and then sent into to the editor's state chart.

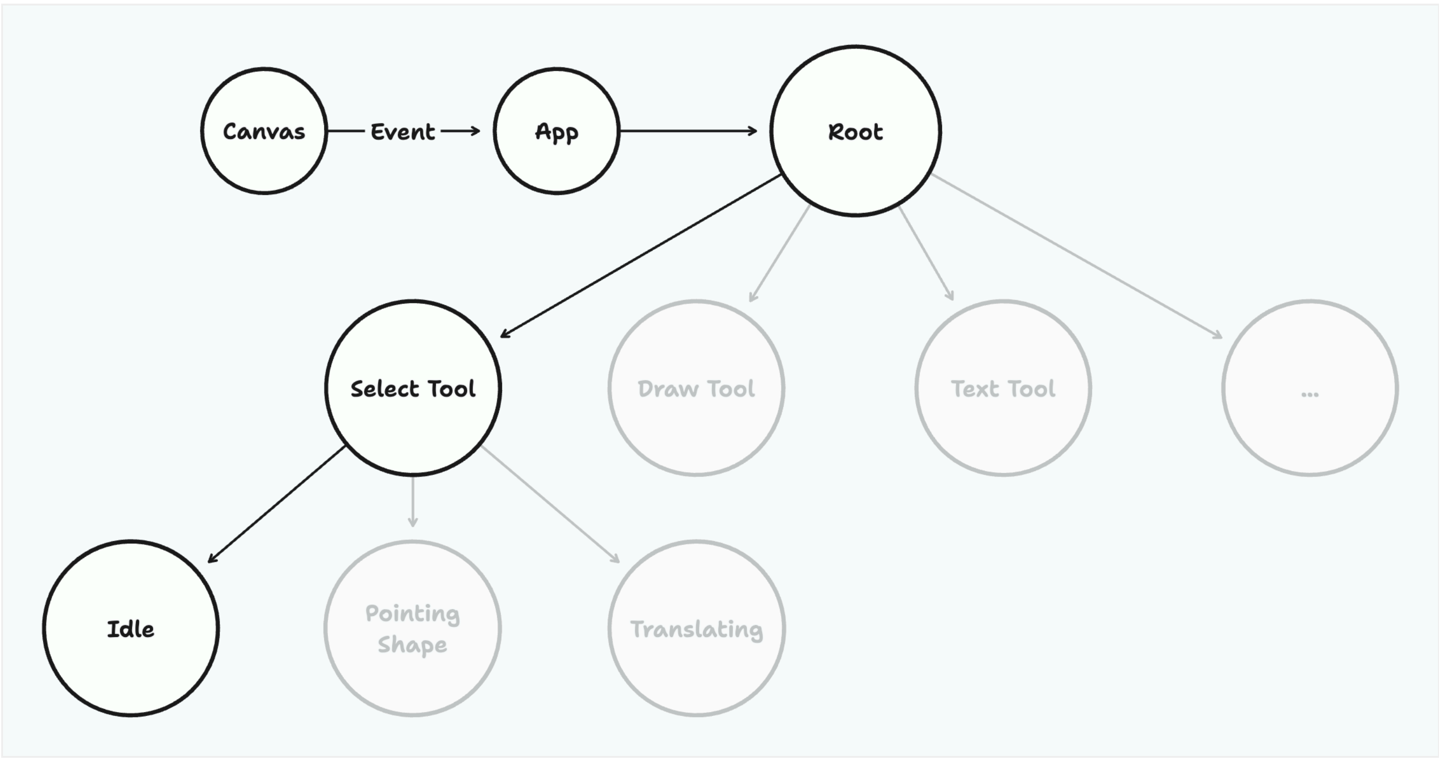
You shouldn't need to use the `Editor.dispatch` method directly, however you may write code in the state chart that responds to these events. See the Tools page to learn how to do that, or read below for a more detailed information about the state chart itself.

State Chart

The `Editor` class has a "state chart", or a tree of `StateNode` instances, that contain the logic for the editor's tools such as the select tool or the draw tool. User interactions such as moving the cursor will produce different changes to the state depending on which nodes are active.

Each node can be active or inactive. Each state node may also have zero or more children. When a state is active, and if the state has children, one (and only one) of its children must also be active. When a state node receives an event from its parent, it has the opportunity to handle the event before passing the event to its active child. The node can handle an event in any way: it can ignore the event, update records in the store, or run a *transition* that changes which states nodes are active.

When a user interaction is sent to the editor via its `Editor.dispatch` method, this event is sent to the editor's root state node (`Editor.root`) and passed then down through the chart's active states until either it reaches a leaf node or until one of those nodes produces a transaction.



Path

You can get the editor's current "path" of active states via `editor.root.path`. In the above example, the value would be `"root.select.idle"`.

You can check whether a path is active via `Editor.isIn`, or else check whether multiple paths are active via `Editor.isInAny`.

```
editor.store.path // 'root.select.idle'

editor.isIn('root.select') // true
editor.isIn('root.select.idle') // true
editor.isIn('root.select.pointing_shape') // false
editor.isInAny('editor.select.idle', 'editor.select.pointing_shape') // true
```

Note that the paths you pass to `Editor.isIn` or `Editor.isInAny` can be the full path or a partial of the start of the path. For example, if the full path is `root.select.idle`, then `Editor.isIn` would return true for the paths `root`, `root.select`, or `root.select.idle`.

If all you're interested in is the state below `root`, there is a convenience method, Editor.getCurrentToolId, that can help with the editor's currently selected tool.

```
import { track, useEditor } from 'tldraw'

export const BubbleToolUi = track(() => {
  const editor = useEditor()

  // Only show the UI if the bubble tool is active
  if (!editor.getCurrentToolId() === 'bubble') return null
  return <div>Creating bubble</div>
})
```

Inputs

The `Editor.inputs` object holds information about the user's current input state, including their cursor position (in page space *and* screen space), which keys are pressed, what their multi-click state is, and whether they are dragging, pointing, pinching, and so on.

Note that the modifier keys include a short delay after being released in order to prevent certain errors when modeling interactions. For example, when a user releases the "Shift" key, `editor.inputs.shiftKey` will remain `true` for another 100 milliseconds or so.

This property is stored as regular data. It is not reactive.

Editor instance state

The `Editor.getInstanceState` method returns settings that relate to each individual instance of the editor. In the case that the user has the same editor open in multiple tabs, or if there are multiple editors on the same page, then each editor will have its own instance state. See the `TLInstance` docs to learn more about the record itself.

User preferences

The editor's user preferences are shared between all instances. See the `TLUserPreferences` docs for more about the user preferences.

Common things to do with the editor

Create a shape id

To create an id for a shape (a `TLShapeId`), use the libary's `createShapeId` helper.

```
import { createShapeId } from 'tldraw'

createShapeId() // `shape:some-random-uuid`
createShapeId('kyle') // `shape:kyle`
```

The `id` property of any record in tldraw is "branded" with the type of that record. For shapes, that means that all shape ids are formatted as `shape:{id}`. The TypeScript type of a record's `id` also includes a reference to the type of the record that it belongs to. TypeScript will complain if you use a regular `shape:some-id` string, but the `createShapeId` helper will provide the type.

Create shapes

To create shapes, use the `Editor.createShape` or `Editor.createShapes` methods.

```
editor.createShapes([
  {
    id,
    type: 'geo',
    x: 0,
    y: 0,
    props: {
      geo: 'rectangle',
      w: 100,
      h: 100,
      dash: 'draw',
      color: 'blue',
      size: 'm',
    },
  },
])
```

A shape must be a partial of the full shape (a `TLShapePartial`). All props are optional except for the `type` of the shape. The shape's corresponding `ShapeUtil` will provide the default props for any props not provided. The `id` will be created if not provided.

Update shapes

To update shapes, use the `Editor.updateShape` or `Editor.updateShapes` methods.

```
editor.updateShapes([
  {
    id: shape.id, // required
    type: shape.type, // required
    x: 100,
    y: 100,
    props: {
      w: 200,
    },
  },
])
```

The update must be a partial of the full shape (a `TLShapePartial`). All props are optional except for the `type` of the shape and its `id`.

Delete shapes

To delete shapes, use the `Editor.deleteShape` or `Editor.deleteShapes` methods.

```
editor.deleteShapes([shape.id])
editor.deleteShapes([shape])
```

You can delete a shape using the shape's `id` or the shape record itself.

Get a shape

You can get a shape with the `Editor.getShape` method.

```
editor.getShape(myShapeId)
editor.getShape(myShape)
```

You can get a shape using the shape's `id` or the shape record itself.

Turn on read only mode

You can use the `Editor.updateInstanceState` method to turn on read only mode.

```
editor.updateInstanceState({ isReadonly: true })
```

Move the camera

You can set the camera to a specific x, y, and zoom with the `Editor.setCamera` method.

```
editor.setCamera(0, 0, 1)
```

Freeze the camera

You can prevent the user from changing the camera using the `Editor.updateInstanceState` method.

```
editor.updateInstanceState({ canMoveCamera: false })
```

Turn on dark mode

You can turn on or off dark mode via the `setUserPreferences` method. Note that this effects all editor instances that share the same user—even instances in other tabs.

```
setUserPreferences({ isDarkMode: true })
```

See the tldraw repository for an example of how to use tldraw's Editor API to control the editor.

Edit this page

Last edited on 22 March 2023

Shapes

Installation