Consolidated Learnings

Key patterns, insights, and lessons learned from building PureScript Tagless D3.

---

1. The MiseEnScene Pattern

Purpose: Package all configuration for a visualization "scene" in one place.

Structure

```
type MiseEnScene = {
    chooseNodes :: (MySimNode -> Boolean)         -- which nodes to show
  , linksShown :: (MyGraphLinkID -> Boolean)      -- which links to render
  , linksActive :: (Datum_ -> Boolean)            -- which links exert
force
  , forceStatuses :: M.Map Label ForceStatus      -- which forces are
active
  , cssClass :: String                            -- top-level CSS class
  , attributes :: MyGraphSceneAttributes          -- visual styling
  , callback :: SelectionAttribute                -- event callback
  , nodeInitializerFunctions :: Array (Array MySimNode -> Array MySimNode)
}
```

Benefits

- **Declarative scene switching**: Just set properties and call `runSimulation`
- **Easy to add new scenes** without touching drawing code
- **Clear separation**: what to show vs. how to show it

Example Usage

```
Scene ClusteredView -> do
  _chooseNodes .= isInMainCluster        -- filter data
  _forceStatuses %= onlyTheseForcesActive ["cluster", "collide"]
  _cssClass .= "clustered"
  _nodeInitializerFunctions .= [positionToGrid]
  runSimulation                           -- apply it all
```

---

2. The runSimulation Pattern

Purpose: Bridge Halogen state changes to D3 visualization updates.

Flow

1. **stageDataFromModel**: Apply filters from scene config to model data
2. **stop**: Stop the running simulation
3. **actualizeForces**: Enable/disable forces per scene config
4. **updateSimulation**: Run D3 General Update Pattern (enter/update/exit)
5. **start**: Restart simulation with fresh alpha

This ensures the simulation always reflects the current scene configuration.

---

3. The datum_ Pattern

Purpose: Type-safe access to D3's opaque `Datum_` type.

Why It's Needed

D3 mutates data (adding x, y, vx, vy fields). We use unsafe coercion to access this data, but isolate it to one place.

Pattern

```
datum_ = {
    id: _.id <<< unboxD3SimNode              -- direct accessor
  , label: _.label <<< unboxD3SimNode
  , x: _.x <<< unboxD3SimNode
  , y: _.y <<< unboxD3SimNode
  , translateNode: \d -> do                   -- computed accessor
      let node = unboxD3SimNode d
      "translate(" <> show node.x <> "," <> show node.y <> ")"
  , nodeRadius: \d -> ...
  , nodeColor: \d -> ...
}
```

Critical Insight

Each visualization needs its own `datum_` accessor object. Don't import from other visualizations - the types will mismatch.

File Organization

```
Viz/YourViz/
├── Files.purs          # Data loading
├── Model.purs          # Types, initializers
├── Model/
│   └── Accessors.purs  # datum_ and link_ accessors
├── Draw.purs           # Update function
└── Attributes.purs     # Visual styling
```

---

4. SimulationM2 Update API

The Wrong Approach

```
-- DON'T: Try to call update from an event handler
onClick = mkEffectFn3 \event datum this -> do
  update { ... }  -- Won't work! Wrong monad!
```

The Correct Pattern

1. **Don't use Effect callbacks at all!** The `update` function is meant to be called from within the monadic drawing context.
2. **Filter the data arrays BEFORE calling update**, not inside an event handler.
3. **Let the update API handle everything** - it will:
   - Merge new filtered data with existing simulation state
   - Swizzle links properly
   - Engage/disengage forces as needed
   - Return enhanced data ready for DOM binding
4. **Use the General Update Pattern** with `updateJoin` to handle enter/update/exit transitions.

Correct Code

```
-- Filter your data and call your drawing function
let filteredNodes = Array.filter predicate allNodes
    filteredLinks = Array.filter linkPredicate allLinks

-- Call update from within the monadic context
enhanced <- update
  { nodes: Just filteredNodes
  , links: Just filteredLinks
  , activeForces: Just activeForces
  , config: Nothing
  , keyFn: keyIsID_
  }

-- Then use updateJoin for the General Update Pattern
node' <- updateJoin nodeGroup Group enhanced.nodes keyIsID_
-- Handle enter/update/exit...
```

Key Insights

1. **The update API is declarative, not imperative**: Describe what data should be shown, it handles the how

2. **Event handlers should be thin**: Just trigger state changes or re-renders, don't manipulate D3 directly
3. **The General Update Pattern is powerful**: enter/update/exit handles all DOM transitions cleanly
4. **Separation of concerns**: Data transformation (pure) → update API (handles simulation) → DOM operations (visual)

---

5. Swizzled vs Unswizzled Links

The Problem

Links change form during simulation initialization:

- **Unswizzled**: `D3Link id r` where `source` and `target` are IDs (String, Int, etc.)
- **Swizzled**: `D3Link_Swizzled` where `source` and `target` are actual node object references

Solution

Distinct foreign types:

```
foreign import data D3Link_Unswizzled :: Type
foreign import data D3Link_Swizzled :: Type

-- The update API signature becomes clearer:
update ::
  { nodes :: Maybe (Array D3_SimulationNode)
  , links :: Maybe (Array D3Link_Unswizzled)  -- Input: IDs only
  , ...
  } ->
  m { nodes :: Array D3_SimulationNode
    , links :: Array D3Link_Swizzled         -- Output: with object
references
    }
```

Benefits

1. **Type safety**: Can't accidentally pass swizzled links where unswizzled are expected
2. **Self-documenting**: Function signatures make it clear what form is needed
3. **Clearer errors**: "Expected D3Link_Unswizzled but got D3Link_Swizzled"

---

6. Extending TreeNode for Hierarchical Data

The Wrong Way

```
-- Creates type cycle
type HierarchicalNode =
  { name :: String
  , children :: Maybe (Array HierarchicalNode)  -- Self-reference!
  }
```

The Right Way

Extend the library's `D3_TreeNode` type using row types:

```
type HierarchicalNodeData =
  { id :: NodeID
  , nodeType :: HierarchyNodeType
  }

type HierarchicalNode = D3_TreeNode (EmbeddedData HierarchicalNodeData)
```

---

7. Link Accessors Pattern

Similar to `datum_`, links need their own accessor object:

```
link_ ::
  { source :: Datum_ -> YourNodeData
  , target :: Datum_ -> YourNodeData
  , linkClass :: Datum_ -> String
  , color :: Datum_ -> String
  }
link_ =
  { source: _.source <<< unboxLink
  , target: _.target <<< unboxLink
  , linkClass: \_ -> "link"
  , color: \d -> linkTypeToColor (unboxLink d)
  }
  where
    unboxLink :: Datum_ -> { source :: YourNodeData, target ::
YourNodeData }
    unboxLink = unsafeCoerce
```

---

8. Bidirectional Events

Halogen to D3

Action handlers update state → runSimulation → updateSimulation

D3 to Halogen

1. Create emitter/listener in Initialize
2. Store callback in scene config

3. D3 click events trigger listener
4. Listener emits Halogen Action
5. handleAction processes it

This creates a clean event loop where both sides can trigger updates.

---

9. Node Initialization Functions

Functions that run on nodes before adding to simulation:

```
-- Position nodes in a grid
positionToGrid :: Array MySimNode -> Array MySimNode
positionToGrid nodes = mapWithIndex positionNode nodes
  where
    columns = ceiling $ sqrt $ toNumber (length nodes)
    positionNode idx (D3SimNode node) =
      let gridPos = numberToGridPoint columns idx
      in D3SimNode node { x = gridPos.x * 100.0, y = gridPos.y * 100.0 }

-- Use in scene:
_nodeInitializerFunctions .= [positionToGrid, unpinAllNodes]
```

---

10. Architecture: Component vs Viz Separation

Component Layer (src/website/Component/)

- Halogen component managing state, UI, and user interactions
- State types and lenses
- Action handlers
- Force library configuration
- HTML rendering

Viz Layer (src/website/Viz/)

- D3 visualization code
- Data models and datum_ accessors
- Drawing functions (initialize, updateSimulation)
- Visual attributes

Benefits

- Clear separation of concerns
- Reusable visualization code
- Testable components
- Easy to add new scenes without touching D3 code

---

11. Force Configuration

Force Library Structure

```
forceLibrary :: Map Label Force
forceLibrary = initialize [
    createForce "charge" (RegularForce ForceManyBody) allNodes [
        F.strength (-300.0)
      , F.distanceMin 1.0
      ]
  , createForce "center" (RegularForce ForceCenter) allNodes [
        F.strength 0.5
      , F.x 0.0
      , F.y 0.0
      ]
  , createLinkForce Nothing [
        F.strength 0.5
      , F.distance 100.0
      ]
  ]
```

Available Force Parameters

From `PSD3.Internal.Simulation.Config`:

- `F.strength`
- `F.radius`
- `F.x`, `F.y`
- `F.distance`
- `F.theta`
- `F.distanceMin`, `F.distanceMax`
- `F.numKey`

Gotcha: Not all D3 force parameters are exposed. Check the library before assuming.

---

12. General Update Pattern (GUP)

The core pattern for data-driven DOM updates:

```
-- Update nodes (Enter-Update-Exit pattern)
node' <- updateJoin node Group enhanced.nodes keyIsID_

-- Enter: create new elements
nodeEnter <- appendTo node'.enter Group [ classed datum_.nodeClass ]
_ <- appendTo nodeEnter Circle attrs.circles
_ <- appendTo nodeEnter Text attrs.labels

-- Exit: remove old elements
```

```
setAttributes node'.exit [ remove ]

-- Update: modify existing elements
setAttributes node'.update updateAttrs

-- Merge enter and update selections for ongoing operations
mergedNodeSelection <- mergeSelections nodeEnter node'.update
```

---

13. State Type Gotchas

Problem

CodeExplorer's State module mixes generic scene infrastructure with Spago-specific types, making it unclear what's reusable.

Solution Needed

- Extract generic scene/state infrastructure to library or shared location
- Leave only visualization-specific types in visualization State module
- Create clear template showing minimal State module structure

What Should Match Library

- `D3SimulationState_`
- `Staging` type
- `D3Selection_`

What Should Be Visualization-Specific

- Your node/link data types
- Scene configuration
- Attributes type

---

14. Troubleshooting

"No selection found": Check that selector in `attach` matches your HTML

Nodes don't move: Check that forces are active and alpha > 0

Links don't connect: Verify key function matches node IDs

Simulation runs forever: Adjust alphaDecay or alphaMin in simulation config

Events don't fire: Check that callback is added to selection attributes

Type errors with update in Effect: You're trying to call update from the wrong monad - use the monadic drawing context

---

Action Items for Library Improvement

- [ ] Rename SimulationM/SimulationM2 to clearer names (e.g., SimulationInit/SimulationUpdate)
- [ ] Add "Common Patterns" cookbook to docs
- [ ] Helper utilities for simulation data manipulation
- [ ] Scaffolding for common event handler patterns
- [ ] Custom type error hints for common mistakes
- [ ] Extract generic State infrastructure from CodeExplorer to library
- [ ] Extend typed approach to SimulationNode and TreeNode