(Hard) Senior Frontend Challenge

Tools needed:

1. Zod (https://github.com/colinhacks/zod)
2. React Flow (https://reactflow.dev)
3. Typescript
4. MongoDB driver for node
5. Node.js

Note: This is a 2 part problem

Context

With React Flow, we're able to build cool and smooth diagramming without having to deal manually with stuff like canvas rendering, edge optimization, data modelling, drag and drop, etc. However, there's still some problems we have to deal with

With trpc, we have seamless type support across frontend and backend code. Coupled with Zod, simple objects declared natively as a zod schema can easily be sent back and forth without manually having to be serialised / deserialised.

With lazy developers who don't care about space complexity, we shove the Node[] exposed by React Flow into MongoDB, which means we have to re-implement Node objects as zod schemas to allow for easy parsing. We've done the following (to save you some time):

```
import { ParseParams, z } from "zod";
import {
  CoordinateExtent,
  HandleElement,
  internalsSymbol,
```

```
  Node as _Node,
  NodeHandleBounds,
  Position,
  XYPosition
} from "react-flow-renderer";
import { ComponentNodeDataType, SubsystemNodeDataType } from "./nodeTypes";

export * from "./nodeTypes";

// helper schemas for the more troublesome interfaces
const _XYPosition: z.ZodType<XYPosition> = z.object({
  x: z.number(),
  y: z.number()
});

const _CoordinateExtent: z.ZodType<CoordinateExtent> = z.tuple([
  z.tuple([z.number(), z.number()]),
  z.tuple([z.number(), z.number()])
]);

const _HandleElement: z.ZodType<HandleElement> = z
  .object({
    id: z.string().nullable().optional(),
    position: z.nativeEnum(Position),
    width: z.number(),
    height: z.number()
  })
  .and(_XYPosition);

const NodeData = z.union([ComponentNodeDataType, SubsystemNodeDataType]);

const _NodeHandleNounds: z.ZodType<NodeHandleBounds> = z.object({
  source: _HandleElement.array().nullable(),
  target: _HandleElement.array().nullable()
});

const partialProps = z.object({
  type: z.string(),
  style: z.record(z.any()),
  className: z.string(),
  targetPosition: z.nativeEnum(Position),
  sourcePosition: z.nativeEnum(Position),
  hidden: z.boolean(),
  selected: z.boolean(),
  dragging: z.boolean(),
  draggable: z.boolean(),
  selectable: z.boolean(),
  connectable: z.boolean(),
  dragHandle: z.string(),
  width: z.number().nullable(),
  height: z.number().nullable(),
  parentNode: z.string(),
  zIndex: z.number(),
  extent: z.literal("parent").or(_CoordinateExtent),
```

```
  expandParent: z.boolean(),
  positionAbsolute: _XYPosition,
  [internalsSymbol]: z
    .object({
      z: z.number(),
      handleBounds: _NodeHandleNounds, // not something we use, no problem left untyped
      isParent: z.boolean()
    })
    .optional()
});

//ComponentNode = regular ol' Node object
//SubsystemNode = container Node, that outline thingy
const NodeData = z.union([ComponentNodeDataType, SubsystemNodeDataType]);

//register the zod schemas for each of the nodes here to get it typed
const NodeData = z.union([ComponentNodeDataType, SubsystemNodeDataType]);
type NodeData = z.infer<typeof NodeData>;

export const Node: z.ZodType<_Node<NodeData>> = z
  .object({
    id: z.string(), // for react-flow-renderer
    position: _XYPosition,
    data: NodeData // this is important later
  })
  .merge(partialProps.partial())

export type Node = z.output<typeof Node>;
```

NodeData consists of two types, ComponentNodeDataType and SubsystemNodeDataType, for the sake of this exercise, what data they contain is arbitrary, but they both contain:

```
_id : ObjectId
```

Where ObjectId comes from “bson”, which you will find in the MongoDB Nodejs driver

In our daily usage, we encounter ObjectId data in two formats: string, or ObjectId object. Since we’re lazy, we don’t want to code in type checks every time we need to access this attribute in both Node types.

The Challenge #1 (Easy)

Implement the _id attribute within both NodeDataTypes such that it’s parse function accepts either string or ObjectId, but the schema object itself only emits ObjectId as both input and output type, e.g.:

```
import yourImplementationNodeDataType from ".../your/file"

type correctType = z.infer<typeof yourNodeDataType>
type correctInputType = z.input<typeof yourNodeDataType>
type correctOutputType = z.output<typeof yourNodeDataType>

// all 3 of them should emit:
// {
//   ...<other arbitrary attributes>...
//   _id?: ObjectId | undefined;
// }


//both invocations should not fail
const actualNodeDataFromString = yourNodeDataType.parse({
  _id: "random hex string here"
})

const actualNodeDataFromObjectId = yourNodeDataType.parse({
  _id: ObjectId.createFromHexString("hex string here")
})

//both invocations should return something that emits
// {
//   ...
//   _id?: ObjectId | undefined;
// }
```

The Bonus Challenge #2 (Hard)

Now that the schema is defined, we turn our focus on our diagram. React Flow’s container node mechanics doesn’t support Container Nodes automatically expanding or shrinking when child nodes are moved around (see it in effect: https://reactflow.dev/docs/examples/nodes/dynamic-grouping/).

Sure, users can manually resize the parent when they feel like It (see: https://reactflow.dev/docs/examples/nodes/node-resizer/). But that makes for terrible

UX, nobody likes to manually resize container nodes , especially when the diagram in question is massive and complex.

Implement dynamic resizing logic into the React Flow lifecycle. By moving children nodes around, their parent node (the container) should be able to dynamically expand / shrink to accommodate it’s children.

Nice to haves:

- Clean, easy to read code
- Organised project structure
- Documentation if neccessary
- Your thought process
- Anything extra (mention in documentation to help us catch it)