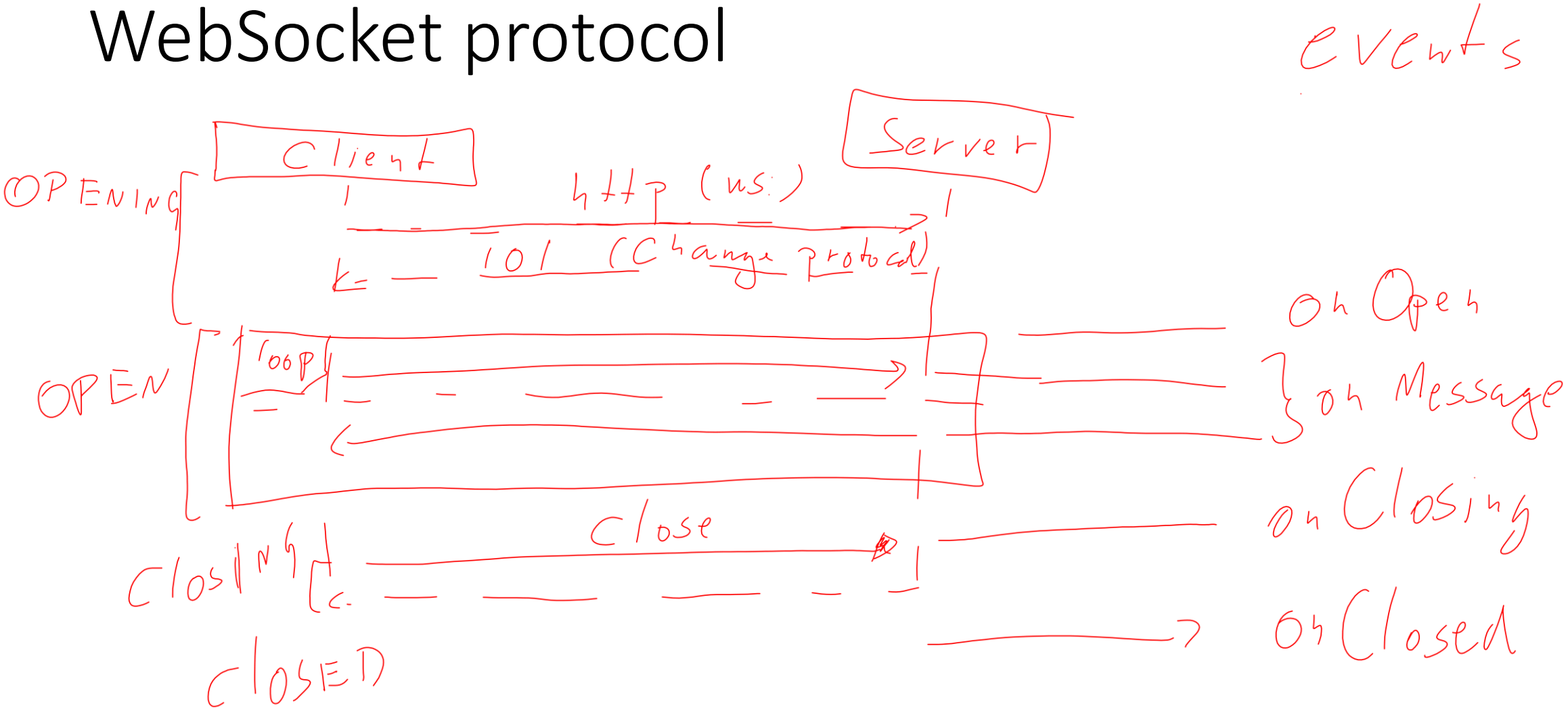
WEB 3, Session 6

Server-client communication

WebSockets

- Like a socket but for HTTP
- Has events open, message, closing, close
- Listening for messages: Not blocking, but with event listeners (callbacks)
- Cannot use promises — can't use async/await
  - Single event

WebSocket protocol



WebSocket server

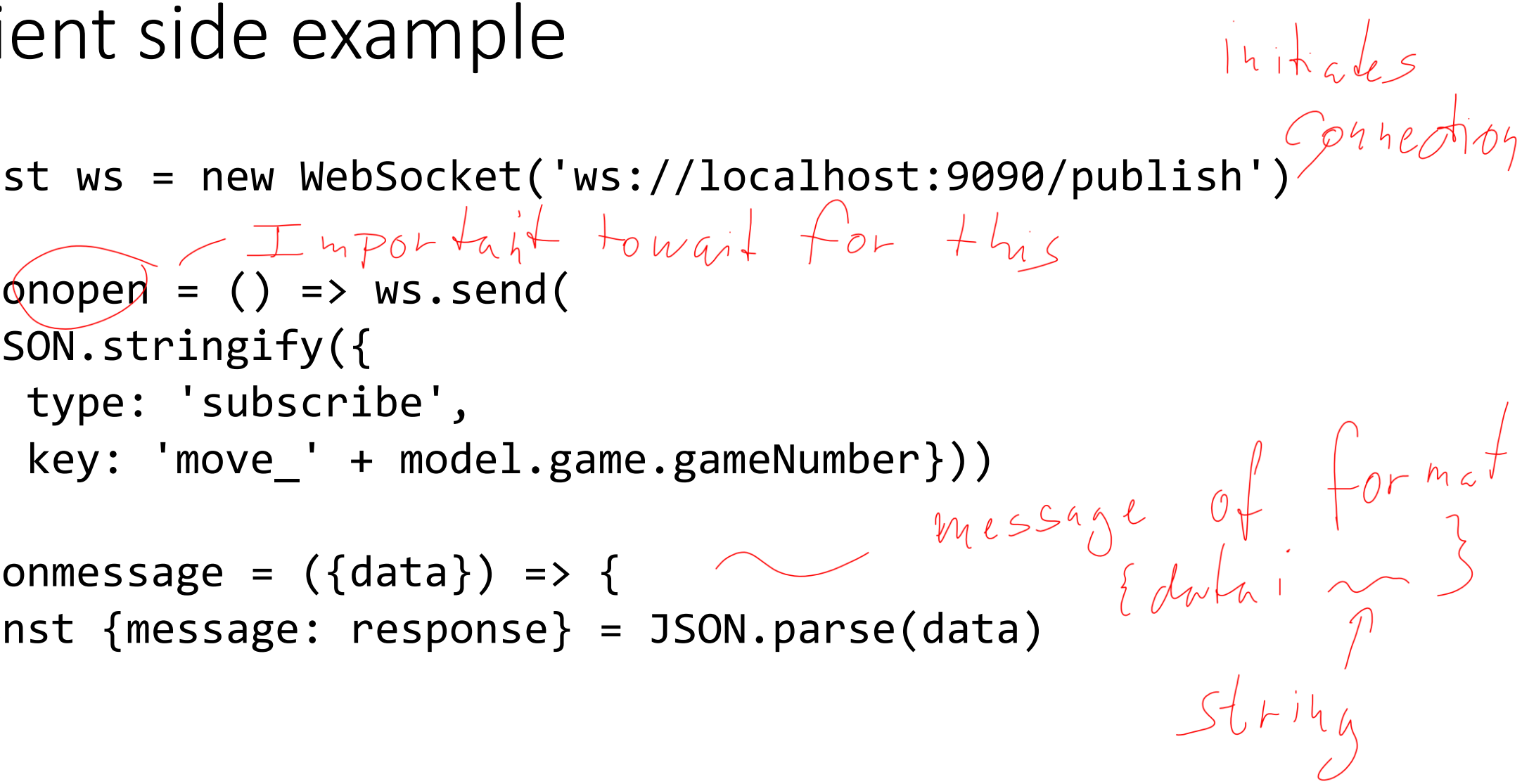
- Need to install 'ws' in your project
- 'ws' has a WebSocketServer
- Remember sockets?
  - The server waits (blocking) for a request
  - The request creates a dedicated socket for the client
- WebSocketServer
  - Listens (non-blocking) for a request
  - The handler is given a dedicated web socket for the client

Code

```
webSocketServer.on('connection', (ws, req) => {
    ws.on('message', message => {
        try {
            const { type, ...params } = JSON.parse(message.toString()) as Command
            if (clients[type] !== undefined)
                clients[type](params, ws)
            else
                console.error(`Incorrect message: '${message}' from ${req.socket.remoteAddress}`)
        } catch (e) {
            console.error(e)
        }
    })
    ws.on('close', () => clients.close({}, ws))
})
```

Client side example

```
const ws = new WebSocket('ws://localhost:9090/publish')

ws.onopen = () => ws.send(
  JSON.stringify({
    type: 'subscribe',
    key: 'move_' + model.game.gameNumber}))

ws.onmessage = ({data}) => {
 const {message: response} = JSON.parse(data)
  …
}
```



Server-sent Events

- Uses HTTP 'keep-alive' connections
- Websocket: Client and server can send to each other as needed
- Server-side events: Server can send to client
- You need a bit more manual handling of responses
  - Send a line* with "data: <the data>"
  - Then send a blank line

* terminated with \n

Headers (server-side)

normal HTTP get

```
gameserver.get('/games/:gameNumber/events', (req, res) => {
  res.setHeader('Connection', 'keep-alive')

  res.setHeader('Content-Type', 'text/event-stream')

  res.setHeader('Cache-Control', 'no-cache')

  ...
```

Sending events

```
res.write(`data: ${JSON.stringify(message)}\n\n`)
```

A line w/ data: ...

blank line

Reacting to connection closing (example)

home_made

Server-Side

```
const sub = subscriptions.subscribe(
  res,
  messageTypeResult.data,
  gameNumberResult.data)

req.on('close', () => {
  subscriptions.unsubscribe(sub.id)
})
```

Client-side

```
const events = new EventSource(
  `http://localhost:8080/games/${gameNumber}/events?type=move`
)

events.onmessage = ({data}) => {
  const message = JSON.parse(data)
  // Handle message here
}

// When done: events.close()
```

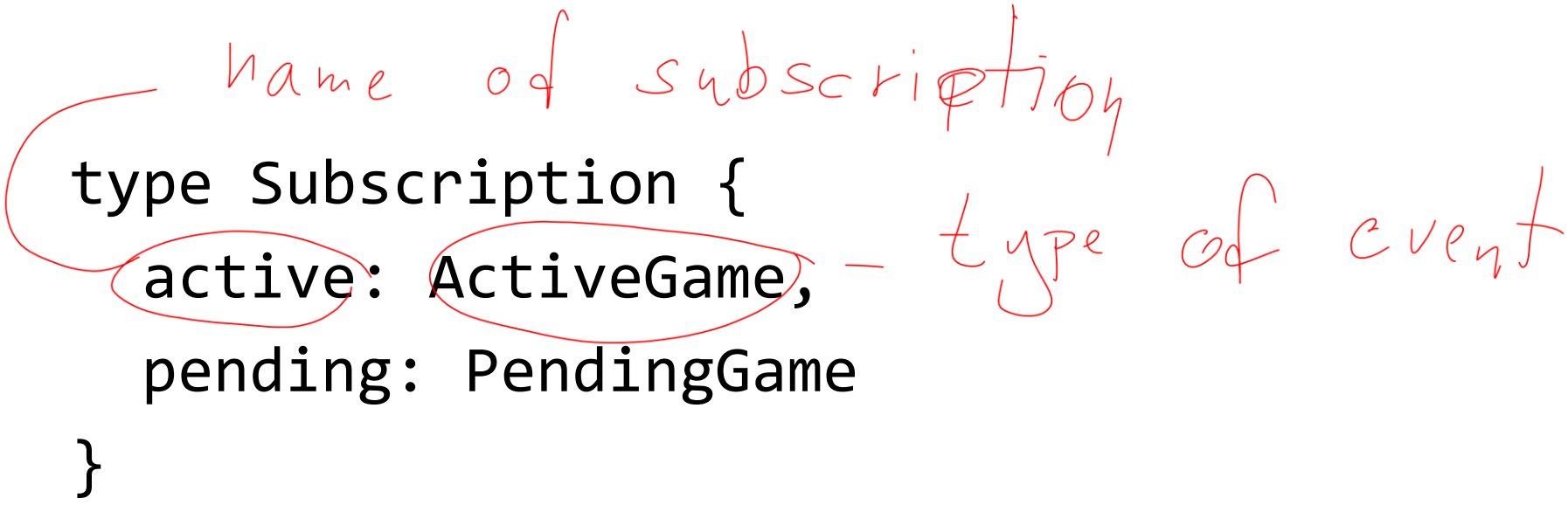
Code

- tic-tac-toe-server-sent-events
  - gameserver.ts
- tic-tac-toe-sse-client
  - Lobby.vue
  - Waiting.vue
  - ActiveGame.vue

GraphQL Subscriptions

- Not implemented in every package
- Is implemented in Apollo Server/Client
- Uses ~~webservices~~ web sockets to communicate
  - we don't have to do much to set ~~webservices~~ up.
- Use a library to manage the communication Web Sockets
- We'll be using PubSub
  - A part of the package 'graphql-subscriptions'
  - Maintains an in-memory queue of messages
  - In production, we would prefer a persistent queue

GraphQL definition (.sdl)



Creating and using PubSub

Same object in resolvers

```
const pubsub: PubSub = new PubSub()
async send(game: PendingGame | IndexedYahtzee) {
  if (game.pending) {
    pubsub.publish('PENDING_UPDATED', {pending: game})
  } else {
    pubsub.publish('ACTIVE_UPDATED', {active: game})
  }
}
const resolvers = create_resolvers(pubsub, api)
```

Resolvers

JavaScript iterator:
{
next(): Object / undefined
}

```
{
…
  Subscription: {
    active: {
      subscribe: () => pubsub.asyncIterableIterator(['ACTIVE_UPDATED'])
    },
    pending: {
      subscribe: () => pubsub.asyncIterableIterator(['PENDING_UPDATED'])
    }
  }
}
```

iterator

Client-side – setting up the client (I)

```
const wsLink = new GraphQLWsLink(createClient({
  url: 'ws://localhost:4000/graphql',
}))


const httpLink = new HttpLink({
  uri: 'http://localhost:4000/graphql'
})
```

Client-side – setting up the client (II)

```
const splitLink = split(
  ({ query }) => {
    const definition = getMainDefinition(query)
    return (
      definition.kind === 'OperationDefinition' &&
      definition.operation === 'subscription'
    )
  },
  wsLink,
  httpLink,
)
```

Client-side – setting up the client (III)

```
const apolloClient = new ApolloClient({
  link: splitLink,
  cache: new InMemoryCache()
})
```

Using the client (I)

local name

```
const gameSubscriptionQuery = gql`subscription GameSubscription {
  active {
    id
    …
    scores {
      slot
      score
    }
  }
}`
```

name of sub

Chosen data

Using the client (II)

```
const gameObservable = apolloClient.subscribe({
  query: gameSubscriptionQuery
})
gameObservable.subscribe({
  next({data}) {
    const game = data.active
    …
  },
  error(err) {
    console.error(err)
  }
})
```

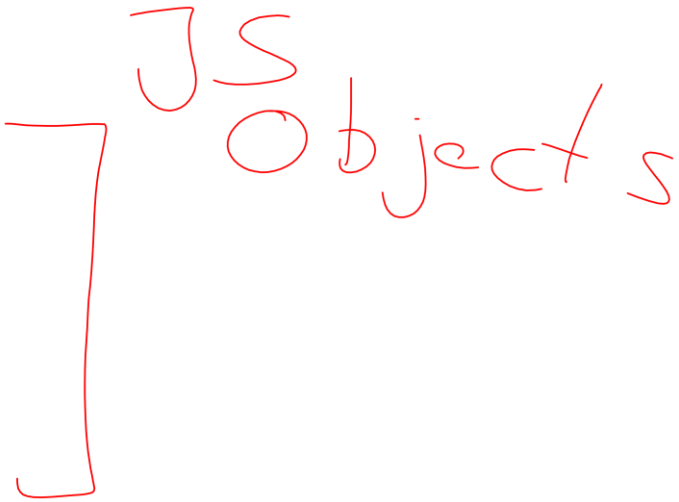
Zod

Why Zod?

- TypeScript types gives us some safety, but they are gone at runtime
- Particularly a problem with
  - User input
  - Data coming in to webservices
- Zod allows us to
  - Validate the data
  - Get data that we know the type of
  - Use typescript types generated by Zod

Defining primitives

```
import * as z from 'zod'


z.number()
z.boolean()
z.string()
…
```



Defining objects

```
z.object({
  conceded: z.literal(false),
  x: z.number(),
  y: z.number(),
  player: Player
})
```

a Zod validation

Similar to

```
typeMove = {
  conceded: false,
  x: number,
  y: number,
  player: Player
}
```

in TS

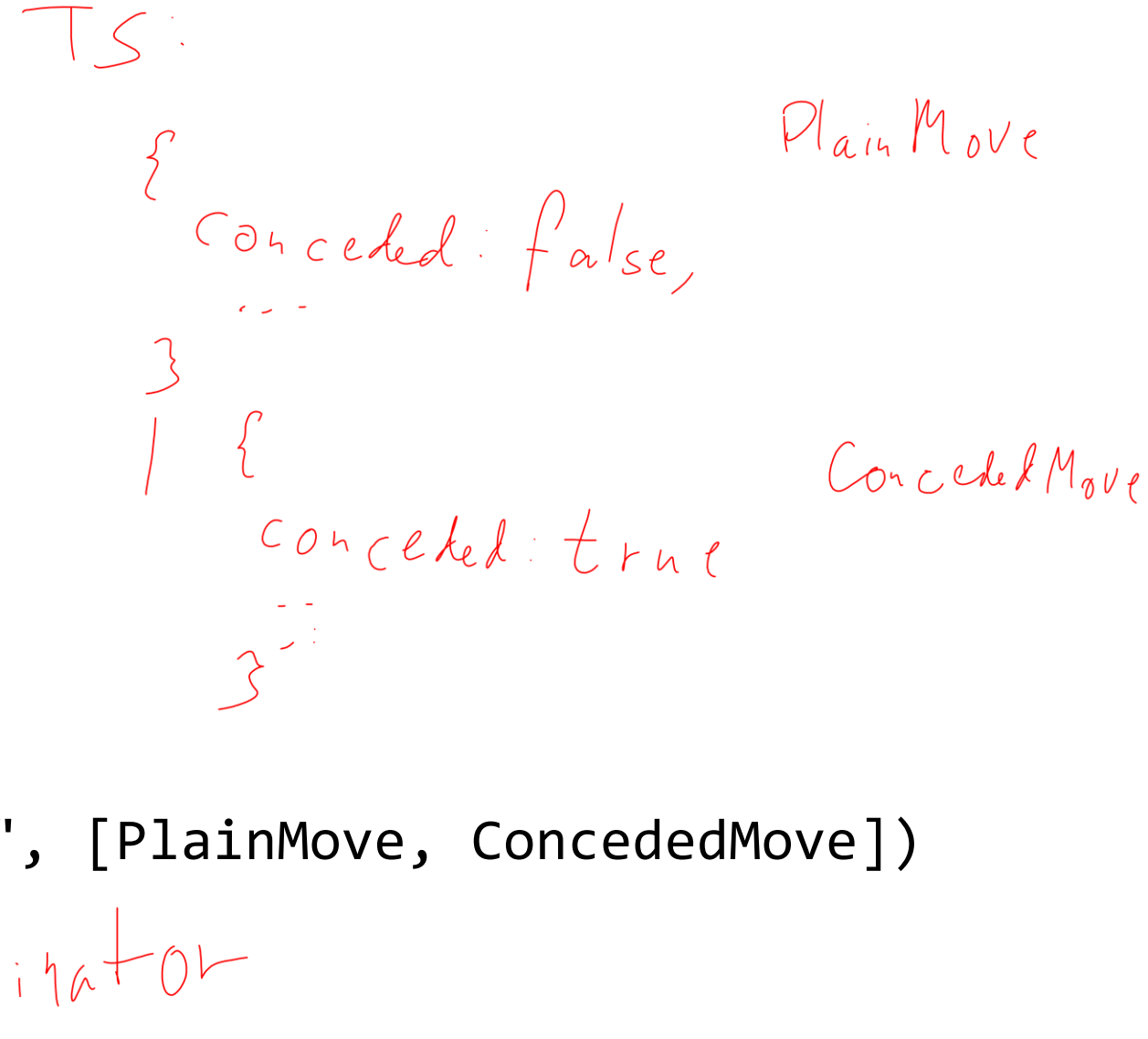
Defining arrays

```
const Row = z.array(
  z.object({
    x: z.number(),
    y: z.number(),
  })
)

const Board = z.array(
  z.array(Tile).length(3)
).length(3)
```

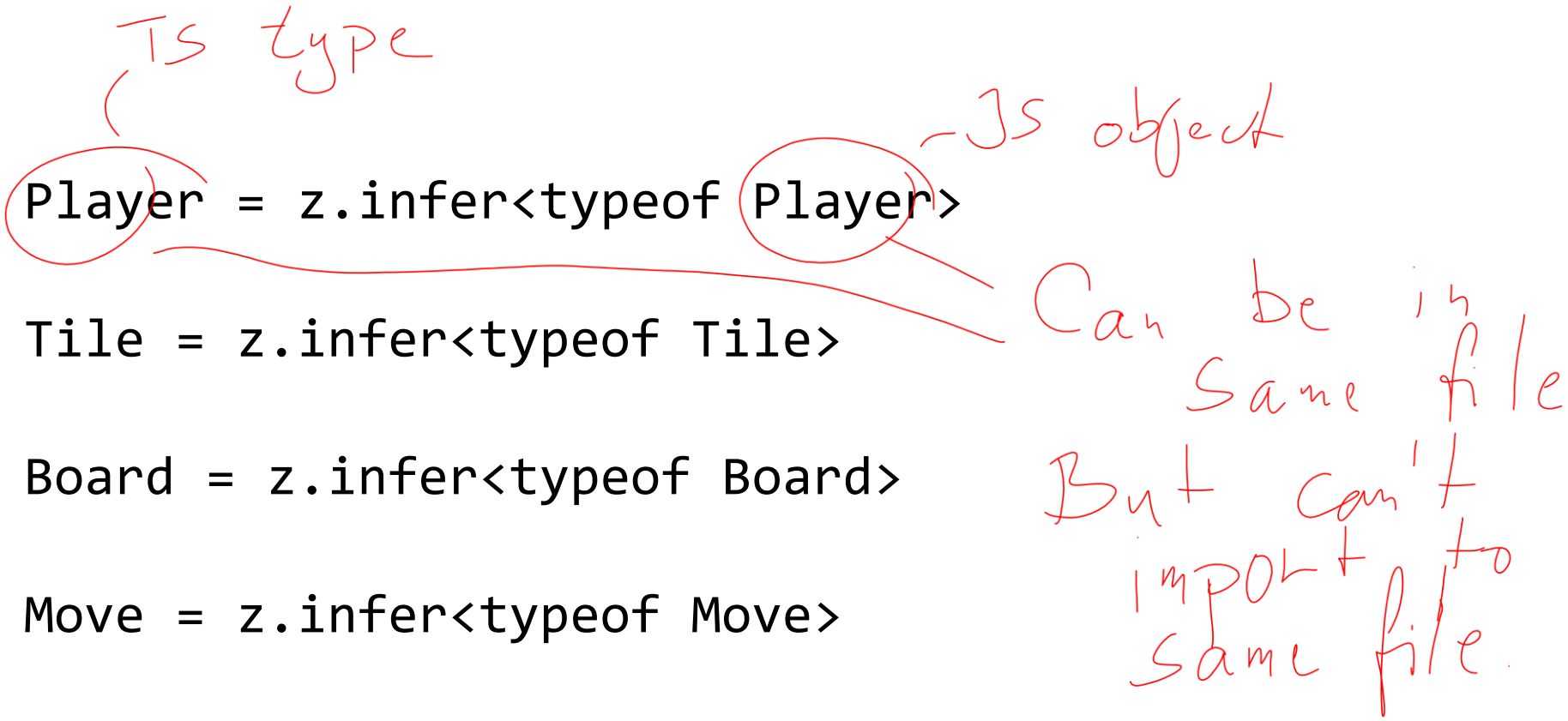
Constraints

Specialities

```
z.enum(['X', 'O'])

z.nullable(Player)

z.literal(true)

z.discriminatedUnion("conceded", [PlainMove, ConcededMove])
```



Types




```
export type Player = z.infer<typeof Player>

export type Tile = z.infer<typeof Tile>

export type Board = z.infer<typeof Board>

export type Move = z.infer<typeof Move>

export type WinState = z.infer<typeof WinState>

export type GameData = z.infer<typeof GameData>
```

Enforcing validation rules

```
z.number().parse(req.body)
```

must be number: 7
not string: "7"

```
z.coerce.number().parse(req.params.gameNumber)
```

either number or string

Returns a number
or throws

Error handling with exceptions

```
try {
  z.coerce.number().parse(…)
} catch (e) {
  …
}
```

Error handling with result pattern

```
const gameNumberResult =
    z.coerce.number().safeParse(req.params.gameNumber)
if (!gameNumberResult.success) {
  res.status(400).send(z.prettifyError(gameNumberResult.error))
  return
}
```

Use gameNumberResult.data ← number

Code

- tic-tac-toe-server-sent-events
  - validation.ts
  - model.ts
  - gameserver.ts