



**THANK YOU  
ORGANIZERS!**

# Ego Slide

- Push buttons
- Type things
- Break stuff
- Wear sweatshirt as cape



# Processes, StateCharts and WorkFlows

...and React?

# React Slide

```
export function Counter(){  
  const [count, setCount] = React.useState(0);  
  return <button onClick={() => setCount(previous => previous + 1)}>  
    Count: {count}  
  </button>  
}
```

# 👉 Italian Restaurant 👈

- Customer is seated to a table
- The customer places his order
- The kitchen cooks the food
- The customer eats the food
- The customer pays

...gotta go Fast!

# /api/process

```
export function postProcess(payload){  
  seatPeople(payload.tableId, payload.seatCount)  
  placeOrder(payload.tableId, payload.menuItemId, payload.amount)  
  const amountDue = askForBill(payload.tableId)  
  removeFromBalance(payload.cartNumber, amountDue)  
  markAsPayed(payload.tableId)  
  sendThankYouEmail(payload.email)  
}
```



A tale of Success

# Modeling software around the Real World

- People arrive and seat
- Choose what to eat (lot of time!)
- The waiter takes the order (and brings it to kitchen)
- The chef cooks (time again!)
- Food is served and people eat (guess what? time!)
- Customer request the bill and pay

# We have been LIED

- Everything takes time to happen
- The waiter waited for us to choose
- The owner waited before placing the bill

Long Running Process

## /api/seat

```
export function postSeat(payload){  
  seatPeople(payload.tableId, payload.seatCount)  
}
```

## /api/order

```
export function postOrder(payload){  
  placeOrder(payload.tableId, payload.menuItemId, payload.amount)  
}
```

## /api/checkout

```
export function postCheckout(payload){  
  const amountDue = askForBill(payload.tableId)  
  removeFromBalance(payload.cartNumber, amountDue)  
  markAsPaid(payload.tableId)  
  sendThankYouEmail(payload.email)  
}
```

# API names

/api/checkout

/api/order

/api/seat

# Execution Flow

```
export function postProcess(/* ... */){  
  seatPeople(tableId, seatCount)  
  placeOrder(tableId, menuItemId, amount)  
  const amountDue = askForBill(tableId)  
  removeFromBalance(cartNumber, amountDue)  
  markAsPaid(tableId)  
  sendThankYouEmail(email)  
}
```

# How the Real World works: interleaved interactions

- Customer A is seated to a table
- Customer B is seated to another table next to Customer A
- The waiter takes Customer A order
- The waiter takes Customer B order
- The kitchen receives both orders and cooks them
- The customers eat the food
- The customers pay only upon finishing their food

Is human interaction the only problem?



## /api/checkout

```
export function postCheckout(payload){  
  const amountDue = askForBill(payload.tableId)  
  removeFromBalance(payload.cartNumber, amountDue)  
  markAsPaid(payload.tableId)  
  sendThankYouEmail(payload.email)  
}
```

# Transactions are a LIE too

```
BEGIN TRANSACTION;  
UPDATE card_balances SET balance = balance - 10 WHERE card_number = 42  
UPDATE orders SET paid = 1 WHERE table_id = 1  
COMMIT TRANSACTION;
```

# And what do we do today?

```
export function postProcess(payload){  
  // ...  
  fetch('http://payment.processor/?card=' + payload.cartNumber +  
    + '&amount=' + amountDue)  
  markAsPayed(payload.tableId) // goes BOOM! 💣  
  sendThankYouEmail(payload.email)  
  // ...  
}
```

# ...is it fixed now?

```
export function postProcess(payload){
  try{
    beginTransaction();
    fetch('http://payment.processor/?card=' + payload.cartNumber +
      + '&amount=' + amountDue)
    markAsPayed(payload.tableId) // goes BOOM! 🤯
    sendThankYouEmail(payload.email)
    commitTransaction();
  }catch(e){
    rollbackTrasaction();
  }
}
```

# Real World Problems

- Customer is seated to a table
- The customer places his order for a steak
- The kitchen finds out it's out of steak
- ...BOOM! 🍷

# Upfront checks aren't an option

What about possible downstream problems?

Long Running Process involve different systems

# User is an external service too

- Input: screen
- Output: key presses and clicks



# Beer slide

- Everyone else had one
- So why should't I
- Please send me free beer
- Man, I love lists.

ACID may exist only locally

...long lived  
transactions?

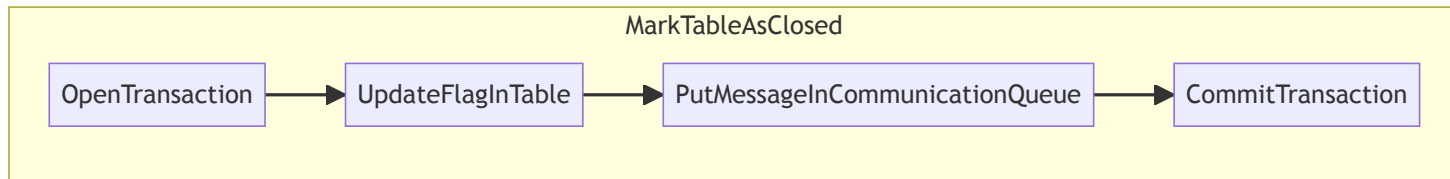
- Sagas!
- in 1987 they had computers!

# /api/checkout



# A communication layer

- Is needed in order to connect each transaction
- Communication becomes part of the transaction as well



# Two kinds of coordination: choreography

```
module Payment
  on "BillRequested"
    processPayment()
    raise event PaymentProcessed()
end module
```

```
module Table
  on "PaymentProcessed"
    closeTable()
    raise event TableClosed()
end module
```

```
module Satisfaction
  on "TableClosed"
    sendThankYouEmail()
    raise event ThankYouSent()
end module
```

# Two kinds of coordination: orchestration

```
module Payment
  on "ProcessPaymentRequest"
    processPayment()
    raise event PaymentProcessed()
end module
```

```
module Table
  on "CloseTableRequest"
    closeTable()
    raise event TableClosed()
end module
```

```
module Satisfaction
  on "SendThankYouEmailRequest"
    sendThankYouEmail()
    raise event ThankYouSent()
end module
```

```
module CloseTableFlow
  raise event ProcessPaymentRequest()
  await PaymentProcessed()
  raise event CloseTableRequest()
  await TableClosed()
  raise event SendThankYouEmailRequest()
end module
```

# Introducing new flows in our application

```
module OfferLiquorFlow
  const liquorId = ProductId(42)
  const [tableId, numberOfGuest] = await event GuestSeated()
  await PaymentRequested()
  raise event PlaceOrder(tableId, liquorId, numberOfGuest)
end module
```



# Events as a whole

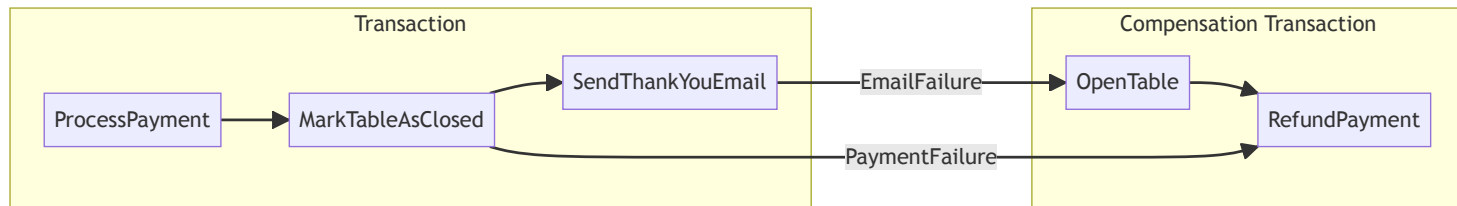
```
module ProductsOrderedReport
  state totalAmount = 0

  on "ProductOrdered"
    totalAmount += event.amount

  on "OrderCancelled"
    totalAmount -= event.amount
end module
```

What happens with failures?

# Compensate all the things!



# Three kind of transactions: Compensable

- Define both the transaction and the compensating transaction
- Should not rely on "just setting the old value back"
- Should be commutative
- WareHouse and Accounting systems are great scenarios were you can find these

# Three kind of transactions: Retriable

- Can't fail but instead may be retried infinitely

# Three kind of transactions: Pivot

- Not compensable in any way
- Once committed, there is no more going back

# Asynchronous checkout in plain JavaScript

```
async function checkout(tableId: number, cardNumber: string, email: string) {  
  const tableDue = await getAmountDue(tableId);  
  await processPayment(cardNumber, tableDue);  
  await markTableAsPaid(tableId);  
  await sendThankYouEmail(email);  
  console.log("Table closed successfully!");  
}
```

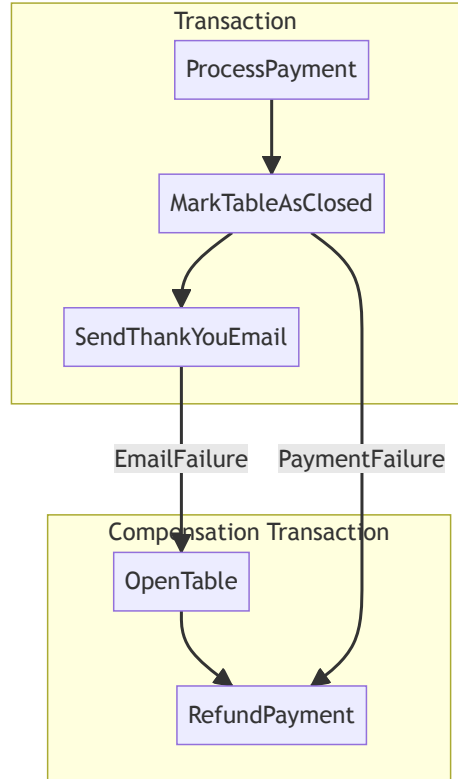
```
export const checkoutDistributed = (  
  tableId: number,  
  cardNumber: string,  
  email: string  
) =>  
  saga(async (step) => {  
    const tableDue =  
      await step(() => getAmountDue(tableId));  
    await step(  
      () => processPayment(cardNumber, tableDue),  
      () => refundAmount(cardNumber, tableDue)  
    );  
    await step(  
      () => markTableAsPaid(tableId),  
      () => openTable(tableId)  
    );  
    await step(() => sendThankYouEmail(email));  
    console.log("Table closed successfully!");  
  });
```



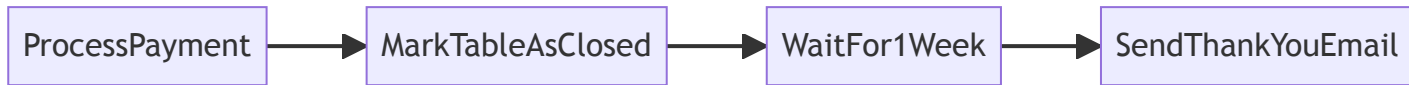
# Simple implementation

```
async function saga<R>(definition: (step: StepFn) => Promise<R>) {  
  const compensations = [];  
  const step = async (transaction, compensate) => {  
    const result = await transaction();  
    if (compensate) compensations.push(compensate);  
    return result;  
  }  
  try {  
    return await definition(step);  
  } catch (e) {  
    for (const compensation of compensations) {  
      await compensation();  
    }  
    throw e;  
  }  
}
```

# Compensation handling



# Waiting is part of the business process



What happens if the process stops?

# Adding a persistence identifier

```
const checkout = (tableId: number, cardNumber: string, email: string) =>
  saga(async (step) => {
    const tableDue = await step("get-amount", () => getAmountDue(tableId));
    await step(
      "payout",
      () => processPayment(cardNumber, tableDue),
      () => refundAmount(cardNumber, tableDue)
    );
    await step(
      "mark-as-paid",
      () => markTableAsPaid(tableId),
      () => openTable(tableId)
    );
    await step("thankyou", () => sendThankYouEmail(email));
    console.log("Table closed successfully!");
  });
```

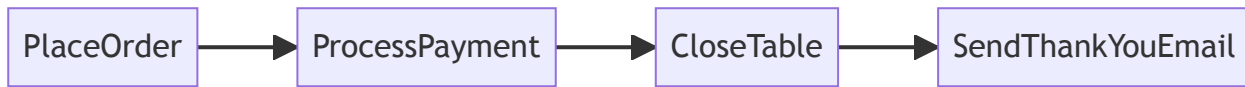
```
const step: StepFn =
  async (persistenceId, transaction, compensate) => {
    if (!persistence.has(persistenceId)) {
      try {
        const result = await transaction();
        persistence.set(persistenceId, { ok: true, result
      } catch (error) {
        persistence.set(persistenceId, { ok: false, error
      }
    }
    const persisted = persistence.get(persistenceId);
    if (persisted.ok) {
      if (compensate) compensations.push(compensate);
      return persisted.result;
    } else {
      throw persisted.error;
    }
  };
```

## ...ACID maybe?

- Atomic: all transactions or compensating transactions will be executed
- Consistent: integrity in each transaction and eventually consistent across services
- Durable: each step commits its internal data
- Isolation: ???

# Lack of isolation

- Potential inconsistent reads
- Potential writes in between other flows

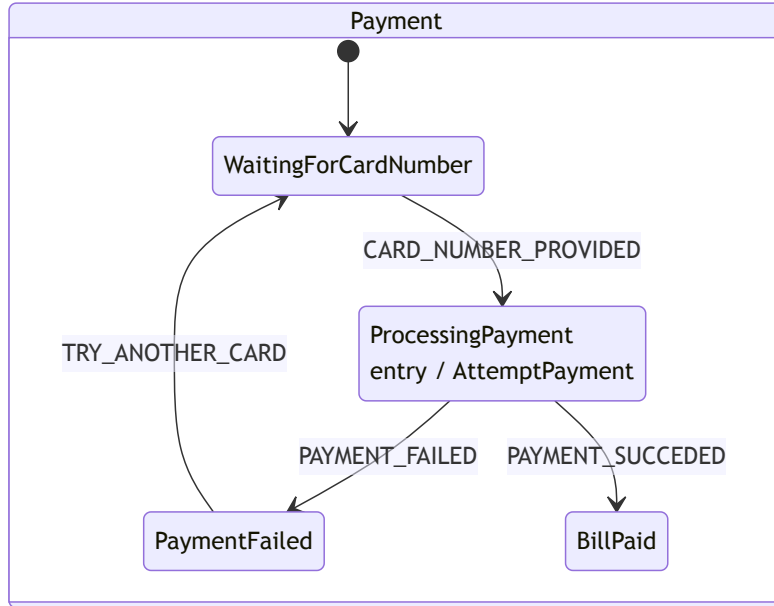




# StateCharts

```
^(state, event) ⇒ state + effects[]^
```

# Payment flow using State Machines



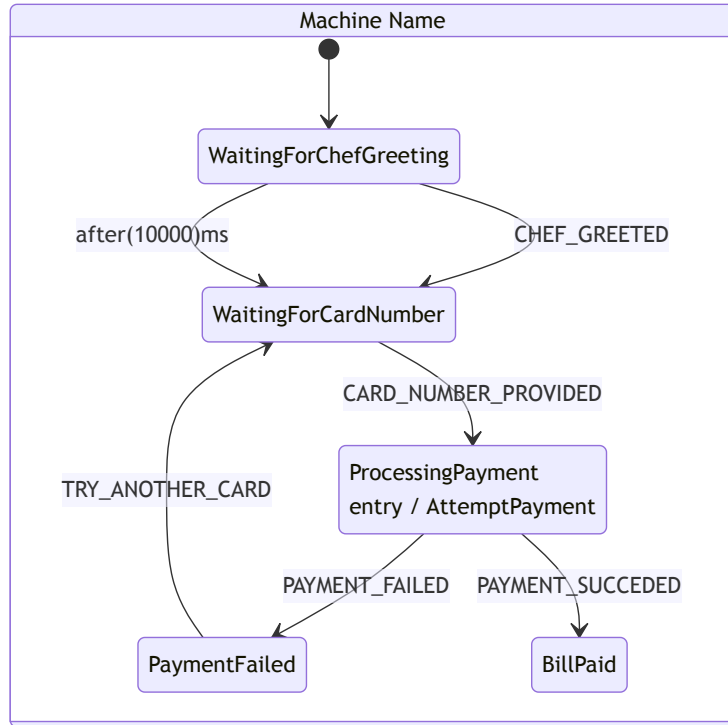
# StateCharts are a DSL rather than Code

- Can be visually shown
- Can be understood by domain experts
- Creates a shared language between devs and experts

# StateCharts are fully serializable

```
{
  "id": "Machine Name",
  "initial": "WaitingForCardNumber",
  "states": {
    "WaitingForCardNumber": {
      "on": {
        "CARD_NUMBER_PROVIDED": {
          "target": "ProcessingPayment"
        }
      }
    },
    "ProcessingPayment": {
      "entry": {
        "type": "AttemptPayment"
      },
      "on": {
        "PAYMENT_FAILED": {
          "target": "PaymentFailed"
        },
        "PAYMENT_SUCCEEDED": {
          "target": "PaymentSucceeded"
        }
      }
    }
  }
}
```

# Evolving flows



# Evolving flows: I do not care!

- Short lived flows
- I Expect to have a consistent UX

# Evolving flows: Restore machine state!

- Work for connections changes or new state added
- What to do when we delete states?

# Evolving flows: Replay events

- Load the new definition with the default state
- Replay all the events that occurred before
- What to do with side effects?
- Works as long your entire machine is deterministic



# Evolving flows:

## Upgrade events

- Do not delete previous state node
- Restore machine state with the new definition
- Trigger upgrade event
- Send new incoming events

# Concurrent Executions

- The waiter comes to the table of 3 people
- The customer order 2 beers and a coke
- The waiter brings the order to kitchen
- Another waiter comes
- The confused customer order 1 beer and 2 coke
- The table receives 2 times the order

# Concurrent execution: Real world

- The boss assigns ranges of tables to each waiter
- Each waiter will pick up only orders of the range of tables it got assigned to

# Concurrent execution: Sharding

- A shard manager will assign range of entities to each worker
- Each worker gets a range of entities to manage and will care only about that
- We ensured that only one worker will process each entity

LRP means... long to reply APIs?

When should we reply to our request? Or should we wait for the completion?

# Long Running Process: The challenges

- *Communication*: they can be orchestration or choreography
- *Distributed transactions*: data may be "dirty" between step
- *Persistent*: because the process may last forever even server restarts
- *Evolving*: because business requirements may change during application lifetime
- *Single executor per workflow*: you do not want concurrency issues right?

# Thanks for your time!

- Twitter/X: @mattiamanzati

