



Patterns for building event-driven web and mobile app backends

Josh Kahn

Tech Leader, Serverless
Principal Solutions Architect
AWS

User interfaces are by their nature event-driven



User interfaces are by their nature event-driven



```
// REACT
export default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click Me!!
    </button>
  );
}
```

But data loading is often synchronous and static

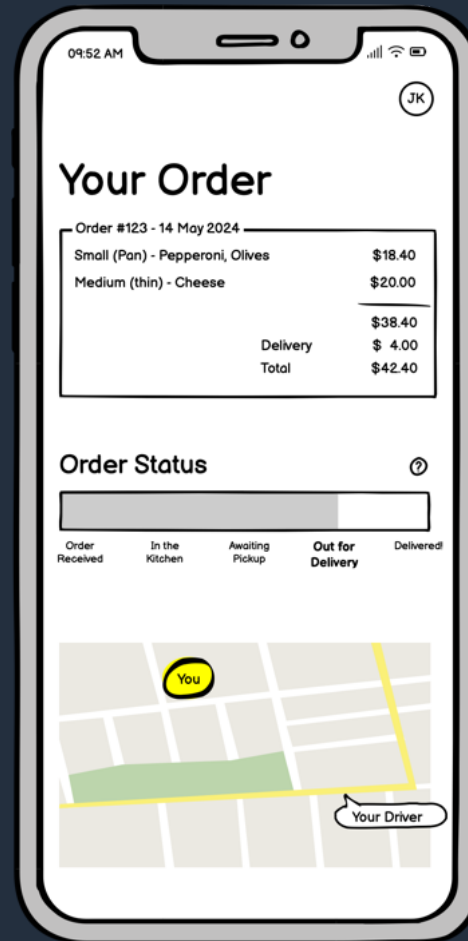


```
const [ data, setData ] = useState(null);

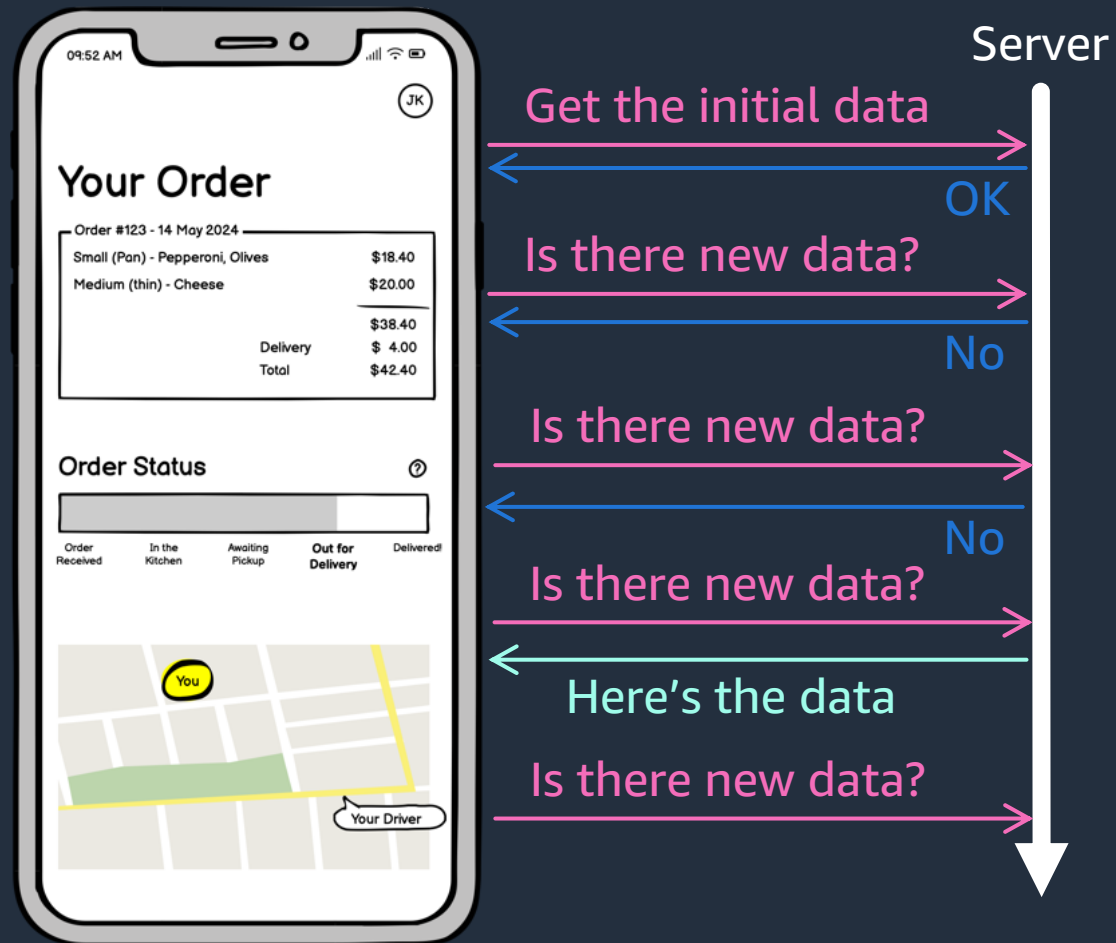
function handleClick {
  fetch("https://api.example.com/stuff")
    .then(resp => resp.json())
    .then(data => setData(data.message))
}
```

Real-time data enhances the experience

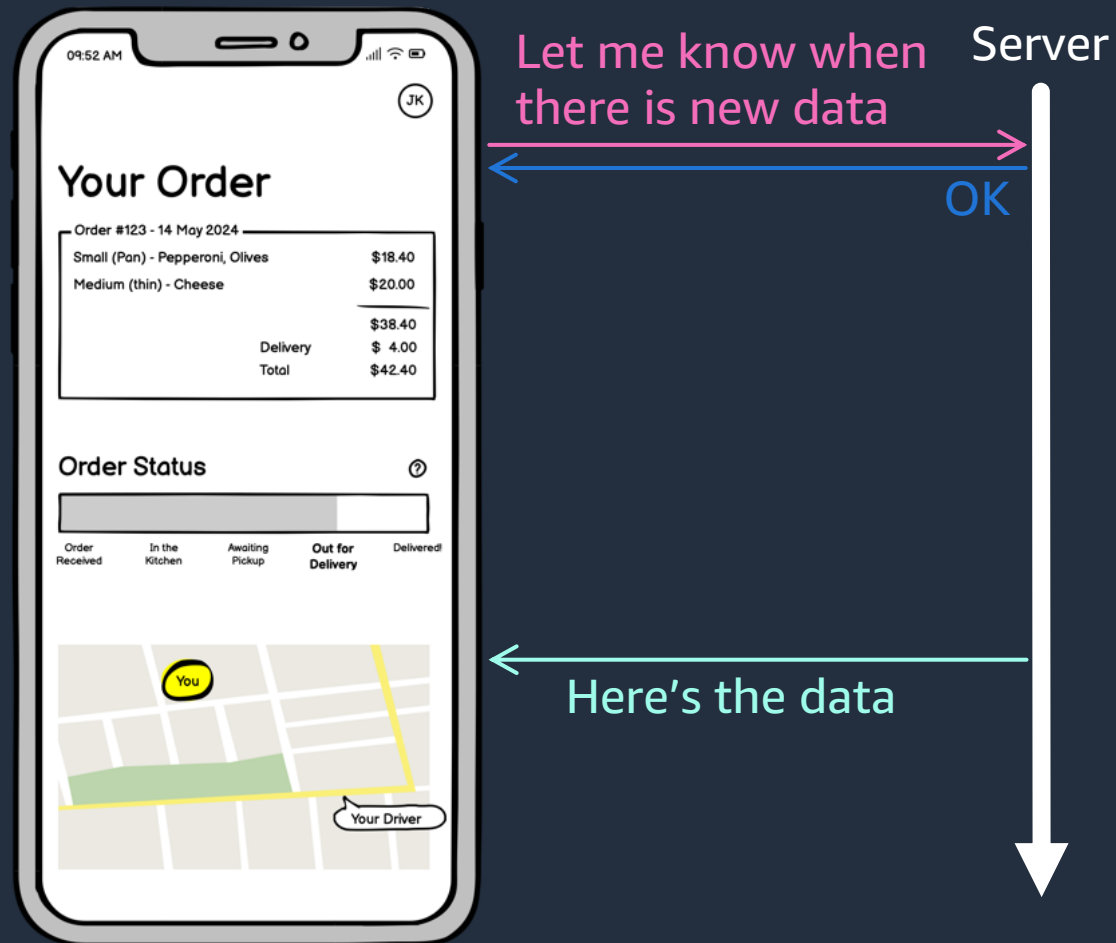
KEEPING USER INTERFACE UP-TO-DATE IS A COLLABORATION WITH THE BACKEND



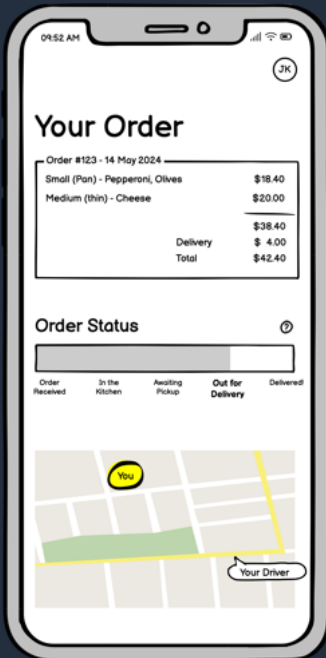
Polling is expensive



Subscription-based approach enables real-time experiences

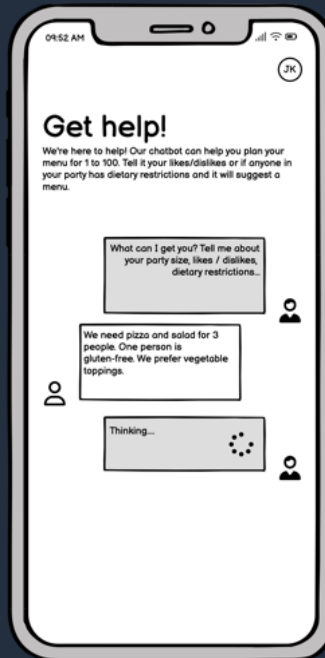


Frontend interaction patterns



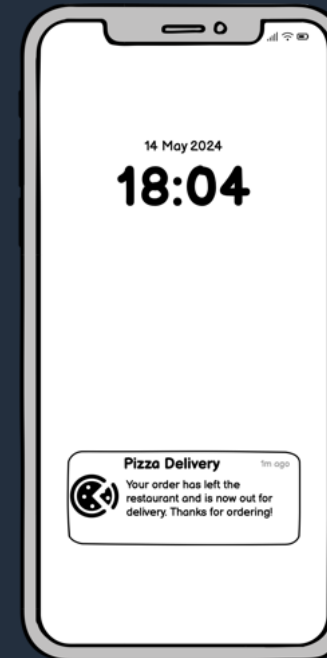
REAL-TIME EVENTS

Something happened, update user interface



LONG RUNNING PROCESS

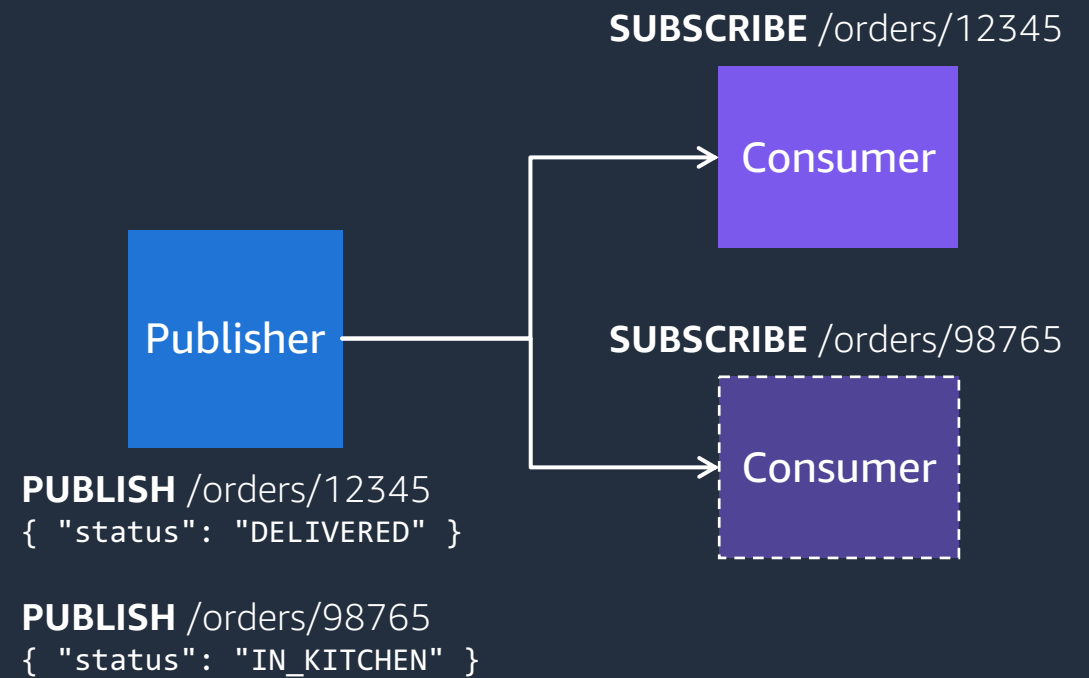
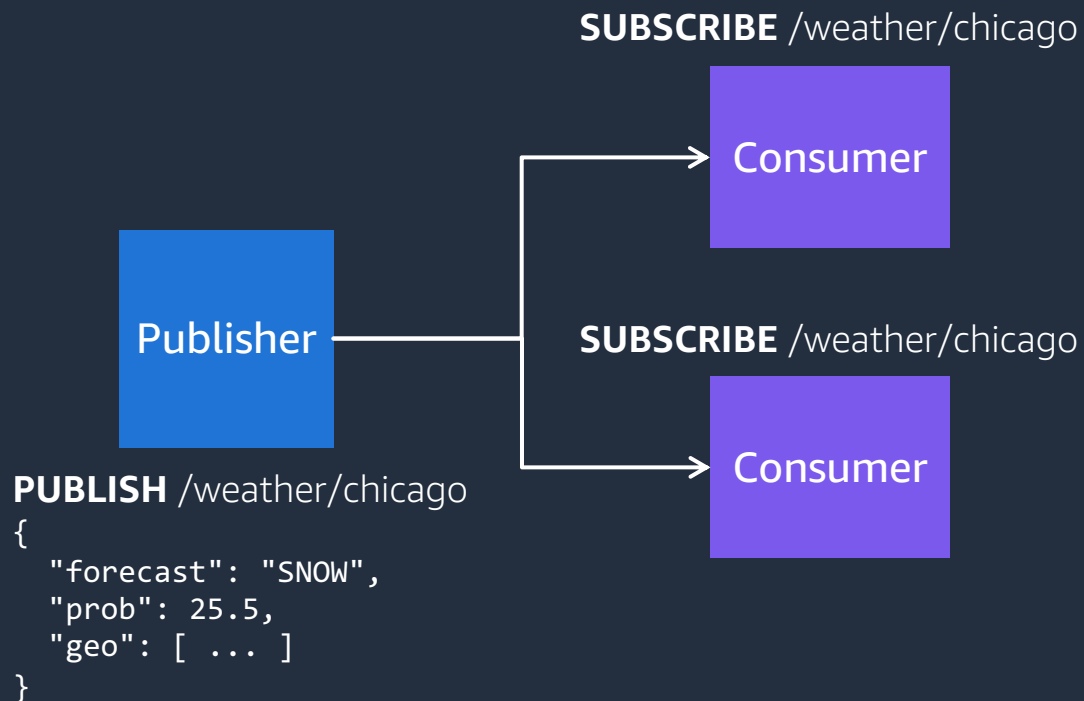
User is waiting for a long-process (30+ seconds)



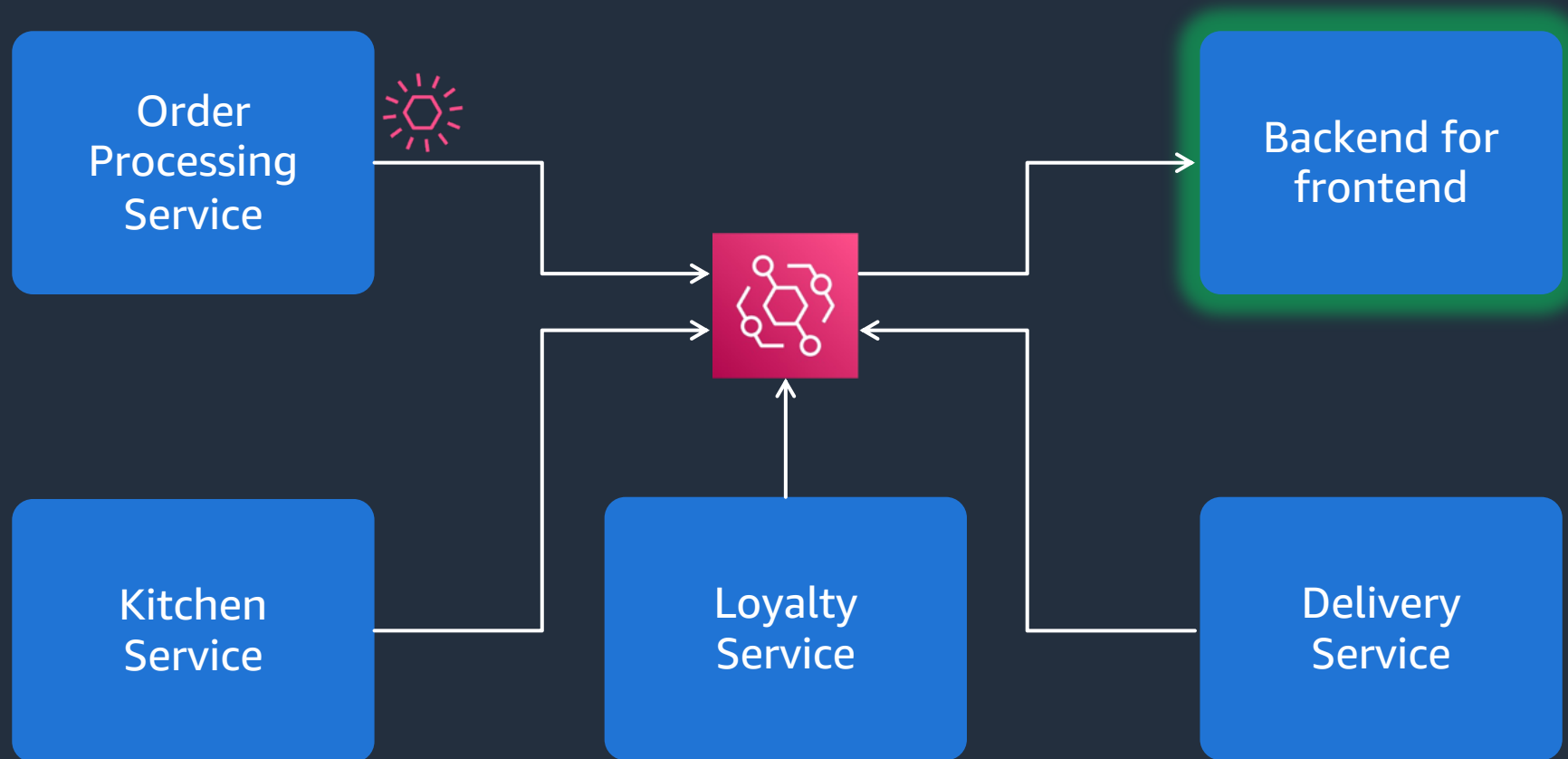
PUSH EVENTS

Something happened, alert the user

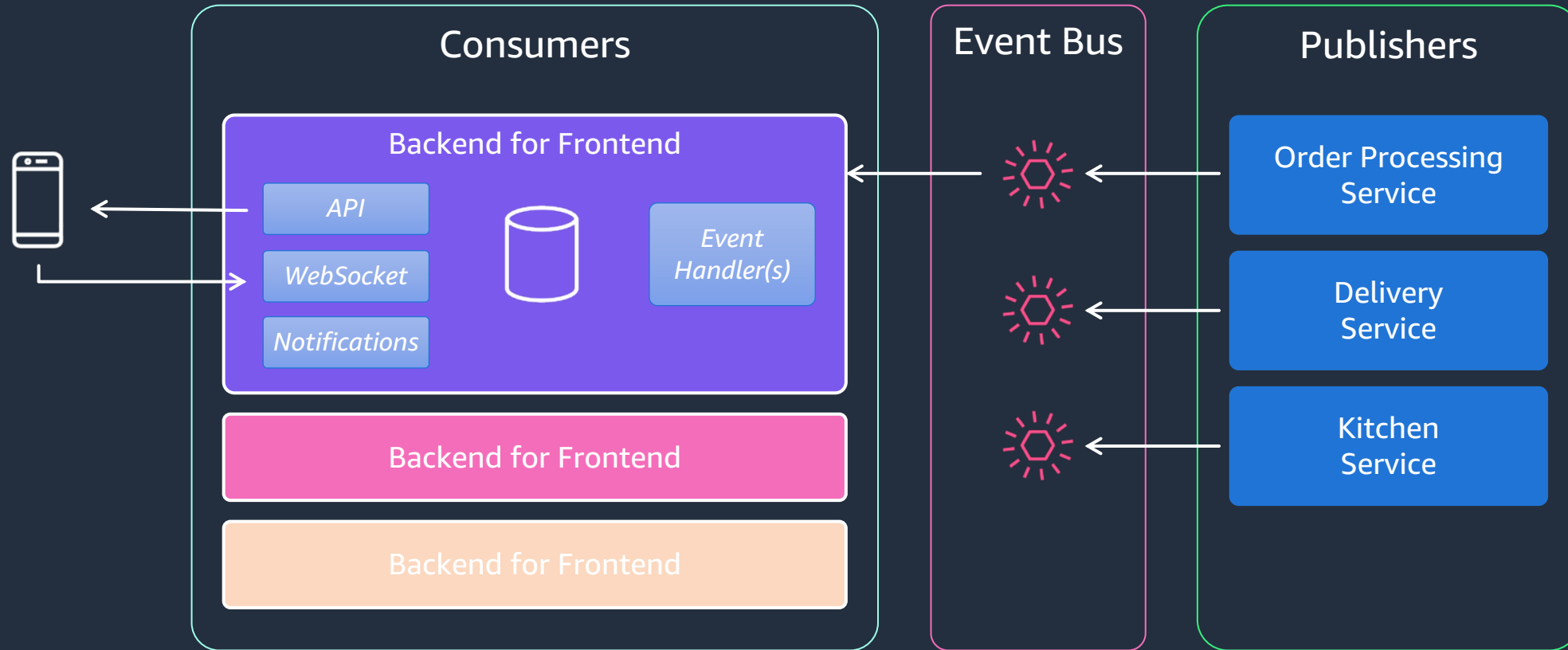
Broadcast & point-to-point patterns



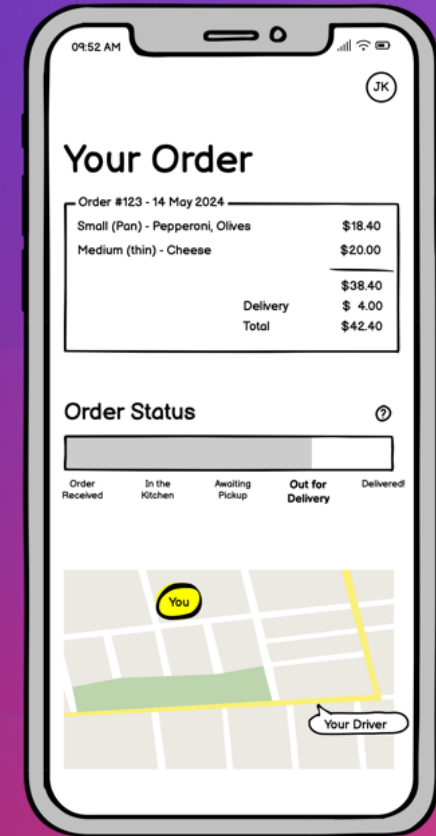
Conceptual architecture: Event-driven food delivery



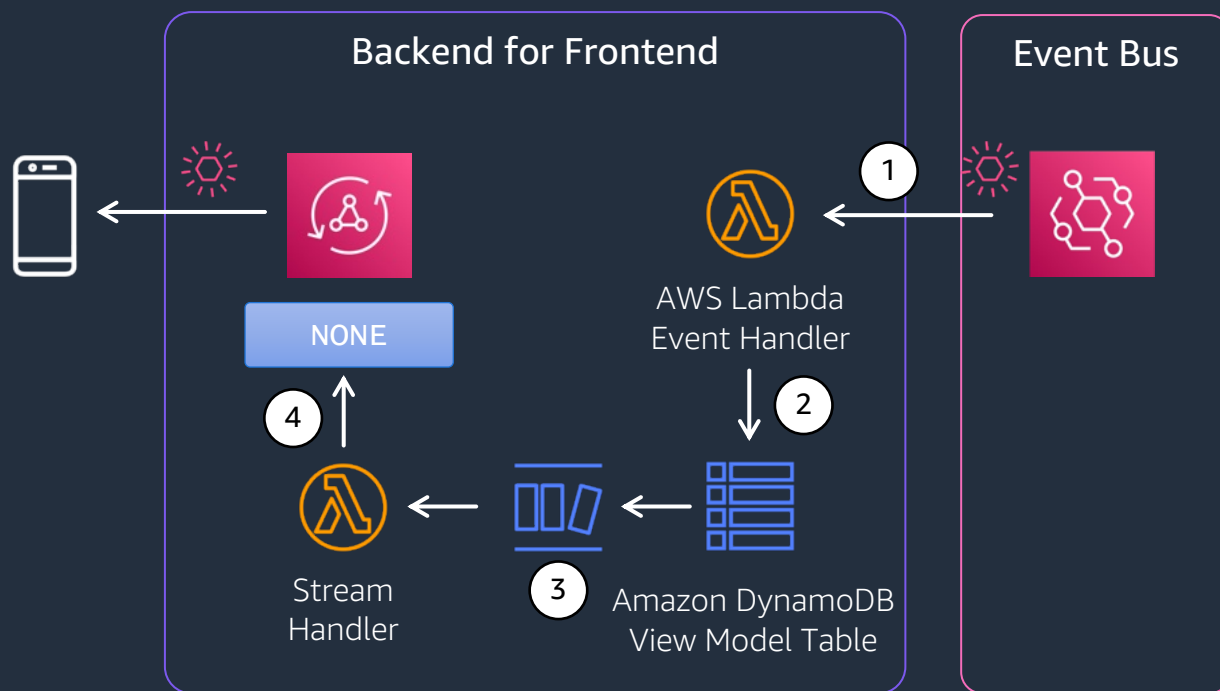
Backend for frontend pattern



Real-time events

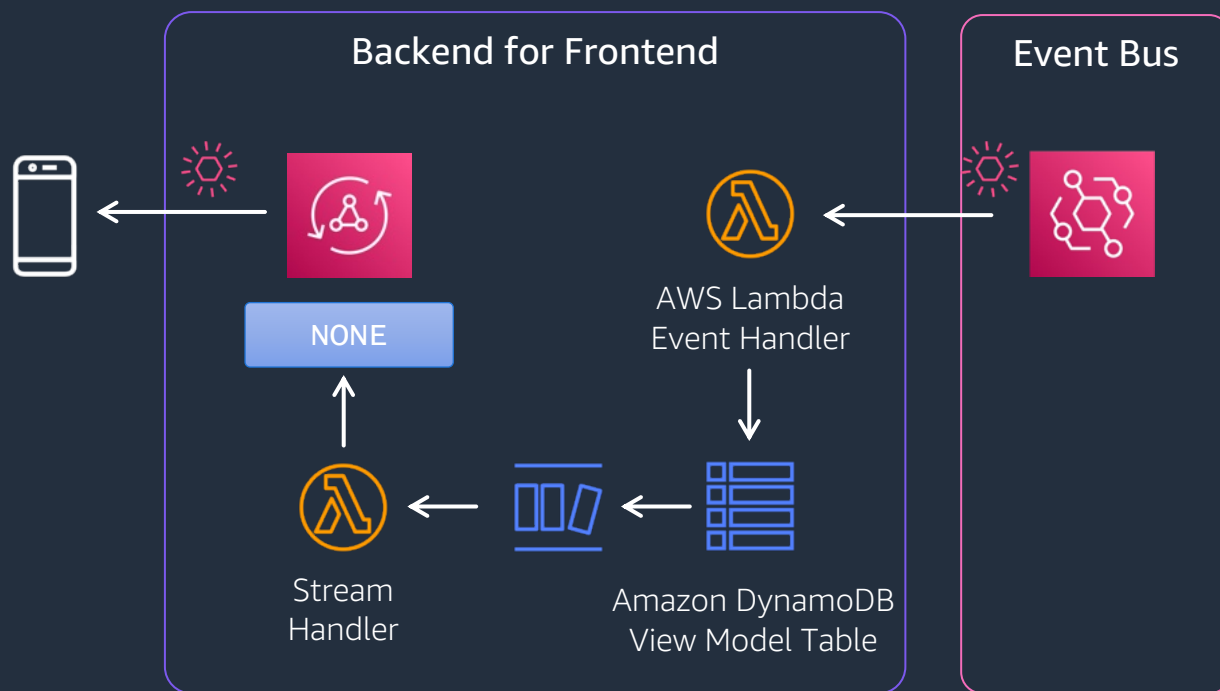


Real-time update w/AWS AppSync



1. Event handler receives, processes event
2. Store in view model table
3. Data change published to stream (filter as needed)
4. Stream handler **mutates via NONE data source**

Real-time update w/AWS AppSync



- + Extensible to **any out-of-band** data change
- + Filter stream as needed
- + Flexible subscriptions
- Additional complexity
- Requires GraphQL
- One-way connection

Auth Options

● Public ● IAM/Cognito ● Lambda/Custom ● OIDC

Broadcast and P2P in AppSync schema

- Schema offers an optional parameter for orderId
 - `onCreateOrUpdateOrder` triggered by both create and update mutations
- Add ! after ID to require parameter:
 - `(orderId: ID!)`

```
type Mutation {  
  createOrder(input: CreateOrderInput!): Order  
  
  updateOrder(input: UpdateOrderInput!): Order  
}  
  
type Subscription {  
  onCreateOrUpdateOrder(orderId: ID): Order  
    @aws_subscribe(mutations: [ "createOrder", "updateOrder" ])  
}
```

Client subscription with AWS AppSync*

Broadcast subscription

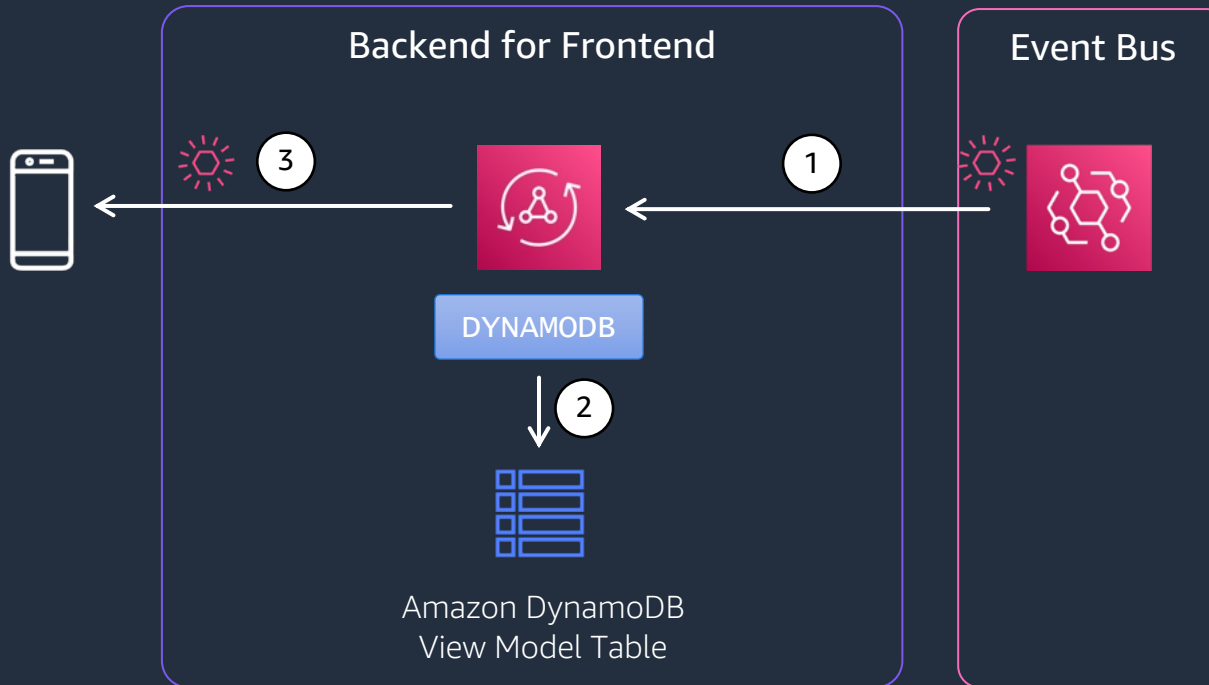
```
const subscription =
  client.subscribe({ query: onCreateOrUpdateOrder })
    .subscribe({
      next: ({ data }) => {
        const { onCreateOrUpdateOrder } = data;
        addUpdateToListing(onCreateOrUpdateOrder);
      },
      error: (error) => console.warn(error)
    });
```

Point-to-point subscription

```
const subscription =
  client.subscribe({
    query: onCreateOrUpdateOrder,
    variables: { orderId }
  })
    .subscribe({
      next: ({ data }) => {
        const { onCreateOrUpdateOrder: { status } } = data;
        setOrderStatus(status);
      },
      error: (error) => console.warn(error)
    });
```

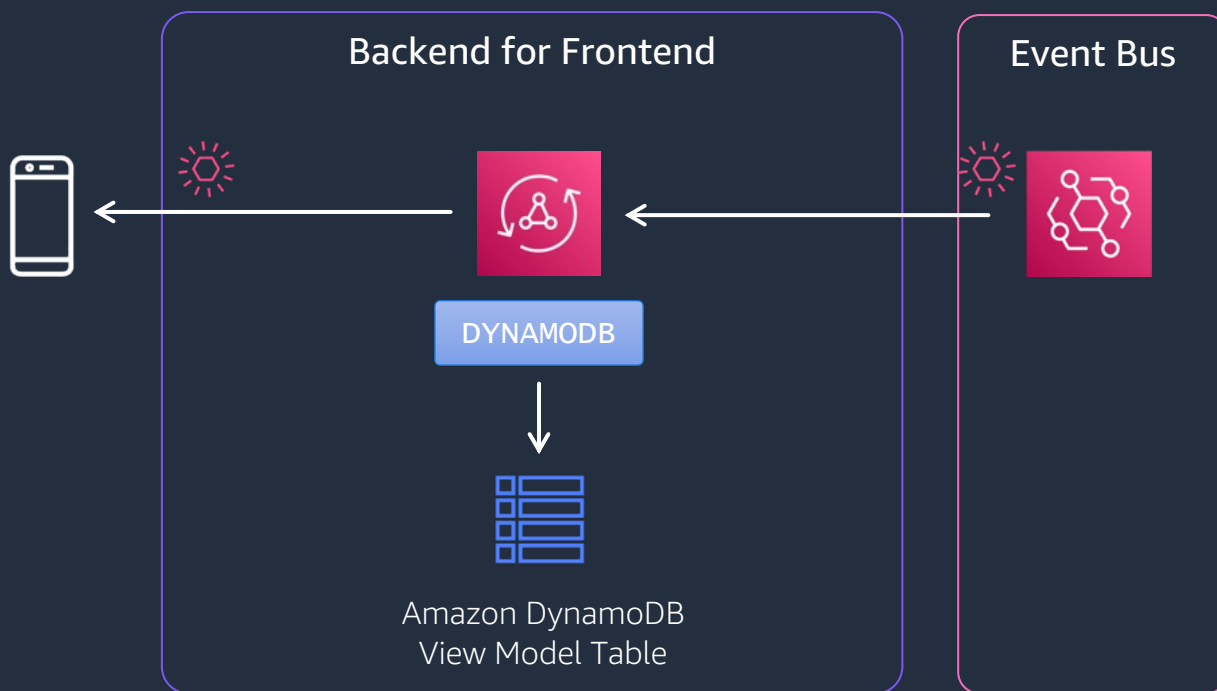
* AWS Amplify API client

Real-time update w/AWS AppSync & Amazon EventBridge



1. Map event payload to GraphQL input
2. Execute mutation via GraphQL endpoint
 - View model update via AppSync resolver
 - Push update to subscribers

Real-time update w/AWS AppSync & Amazon EventBridge



- + Simple, direct integration
- + Mutate data, update subscribers in single step
- + Subscription flexibility
- + Supports private connectivity
- Input transform limited* to EventBridge options
- GraphQL, one-way

Auth Options

● Public ● IAM/Cognito ● Lambda/Custom ● OIDC

Pattern #2: Integrate EventBridge and AWS AppSync



Target 1

Target types

Select an EventBridge event bus, EventBridge API destination (SaaS partner), or another AWS service as a target.

- ☐ EventBridge event bus
- ☐ EventBridge API destination
- ☒ AWS service

Select a target [Info](#)

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)

AppSync

API

Select the API that will receive the event from EventBridge. IAM role needs to be configured as an authorization mode to use as a target.

OrdersGraphQLApi

GraphQL Endpoint ARN

ARN of GraphQL endpoint URL to invoke as a target.

arn:aws:appsync:us-east-2: [REDACTED]

GraphQL API mutation

Select a mutation to invoke when an event matches your event pattern or when schedule is triggered. [View mutations in AppSync](#)

CreateOrder

Configure target input [Info](#)

You can customize the text from an event before EventBridge passes the event to the target of a rule.

Input transformer

Configure input transformer

Selection set

Edit selection set you want to put in the response.

Editor

```
1 {
2   currency
3   customerId
4   items {
5     itemId
6     description
7     quantity
8     unitPrice
9   }
10  orderId
11  numItems
12  status
13  total
14 }
15
```

GraphQL API mutation preview

```
mutation CreateOrder($input: CreateOrderInput!) {
  createOrder(input: $input) {
    currency
    customerId
    items {
      itemId
      description
      quantity
      unitPrice
    }
    orderId
    numItems
    status
    total
  }
}
```

GraphQL is valid

Pattern #2: CloudFormation definition

```
Targets:
- Arn: !GetAtt OrdersGraphqlApi.GraphQLEndpointArn
  RoleArn: !GetAtt OrderApiRuleRole.Arn
  AppSyncParameters:
    GraphQLOperation: >-
      mutation CreateOrder($input: CreateOrderInput!) {
        createOrder(input: $input) {
          customerId
          items {
            itemId
          }
          ...
        }
      }
    ...
  ...
```

Pattern #2: Input transformer magic



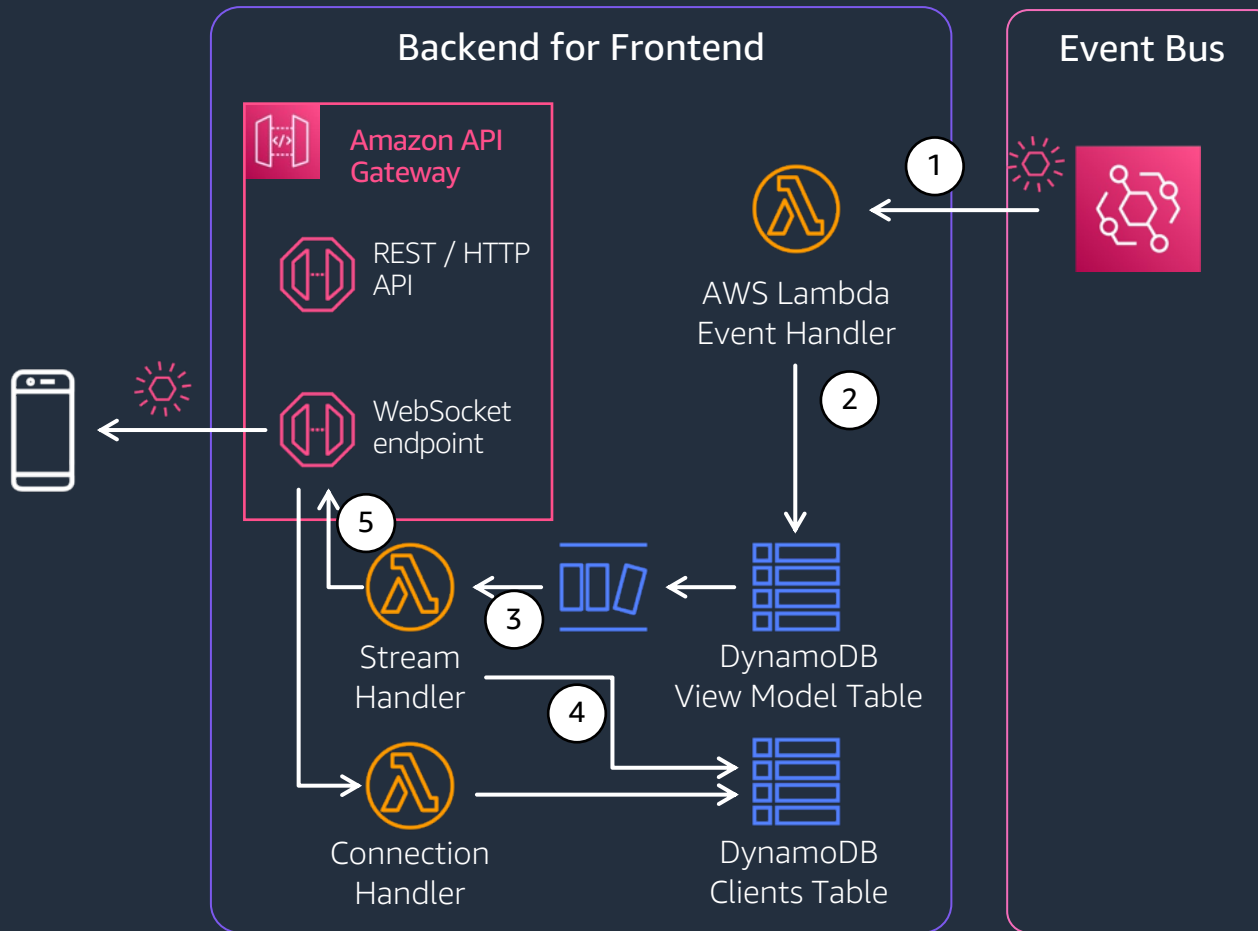
```
Targets:
- Arn: !GetAtt OrdersGraphqlApi.GraphQLEndpointArn
...
InputTransformer:
  InputPathsMap:
    customerId: $.detail.customerId
    ...
  InputTemplate: |-
    {
      "input": {
        "customerId": "<customerId>",
        "items": <items>,
        ...
      }
    }
```

Brief word on mutation selection sets

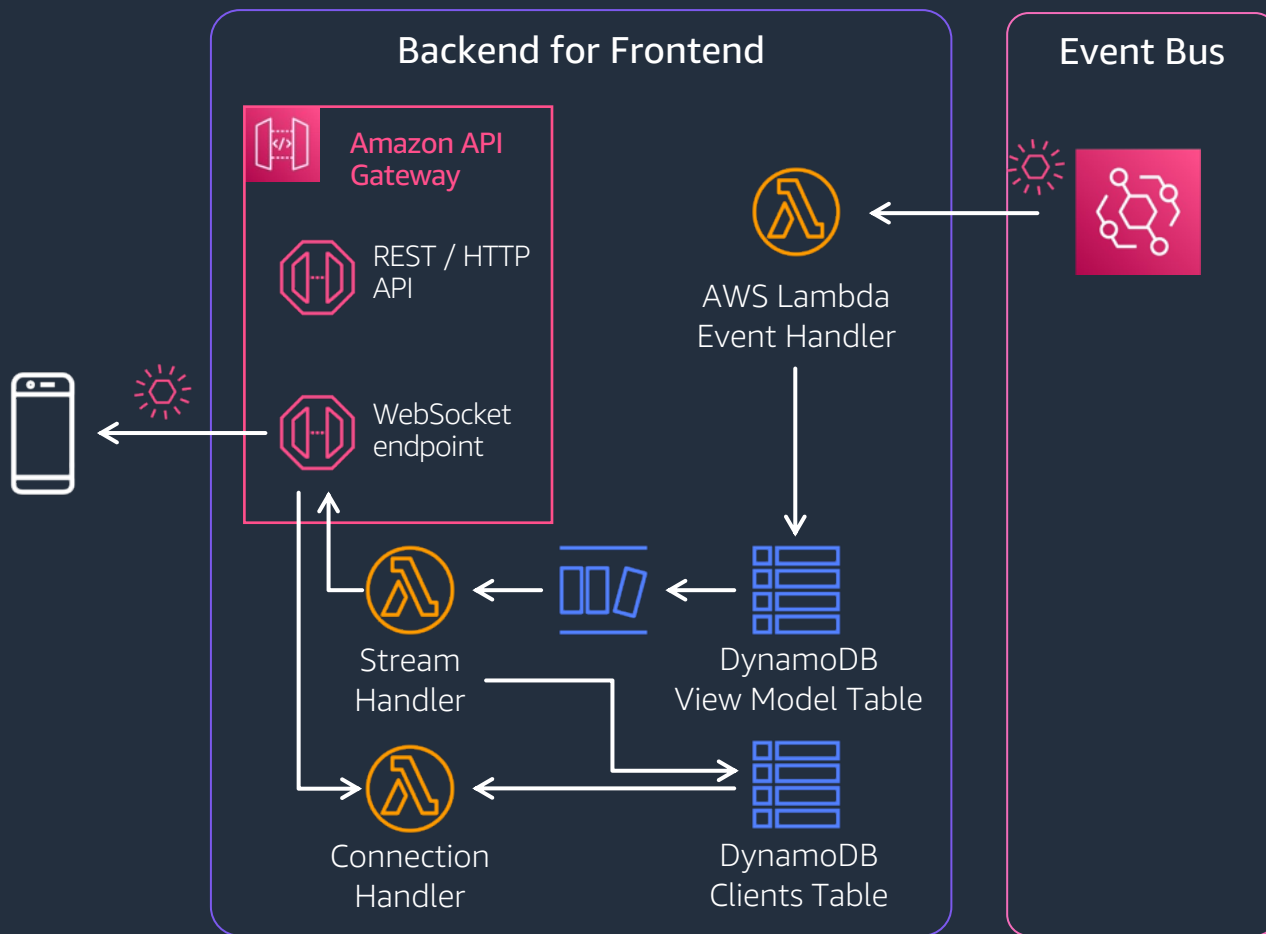
- Mutation **selection set matters**
 - Defines fields returned in response to operation
 - Even in fire-and-forget scenarios
- **AND** fields **available to subscribers**
 - Other fields will be null
 - Tip: include all potential fields of interest for subscribers

```
mutation CreateOrder($input: CreateOrderInput!) {  
  createOrder(input: $input) {  
    currency  
    customerId  
    createdAt  
    updatedAt  
    items {  
      itemId  
      description  
      quantity  
      unitPrice  
    }  
    orderId  
    numItems  
    status  
    total  
  }  
}
```

Real-time update w/Amazon API Gateway



Real-time update w/Amazon API Gateway

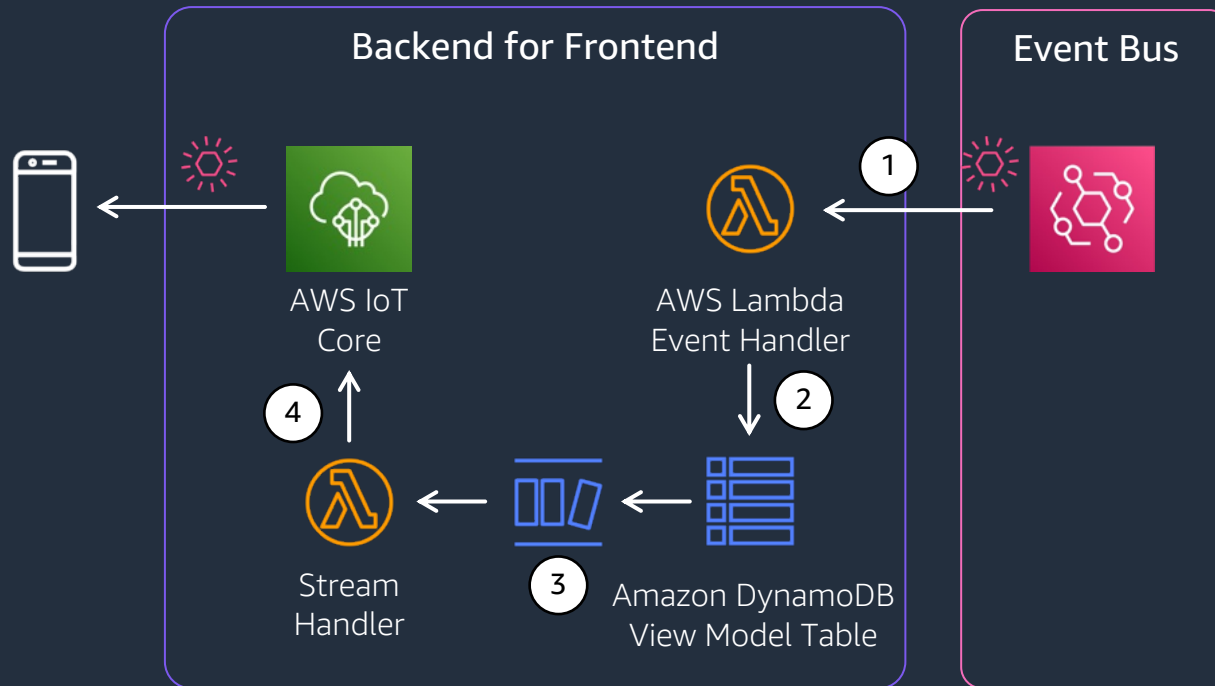


- + Extensible to variety of events
- + Two-way connection
- + Vanilla WebSocket
- DIY connection management
- Messages sent *one by one*, no broadcast
- Multiple API endpoints
- 2-hour timeout, no auto reconnect

Auth Options

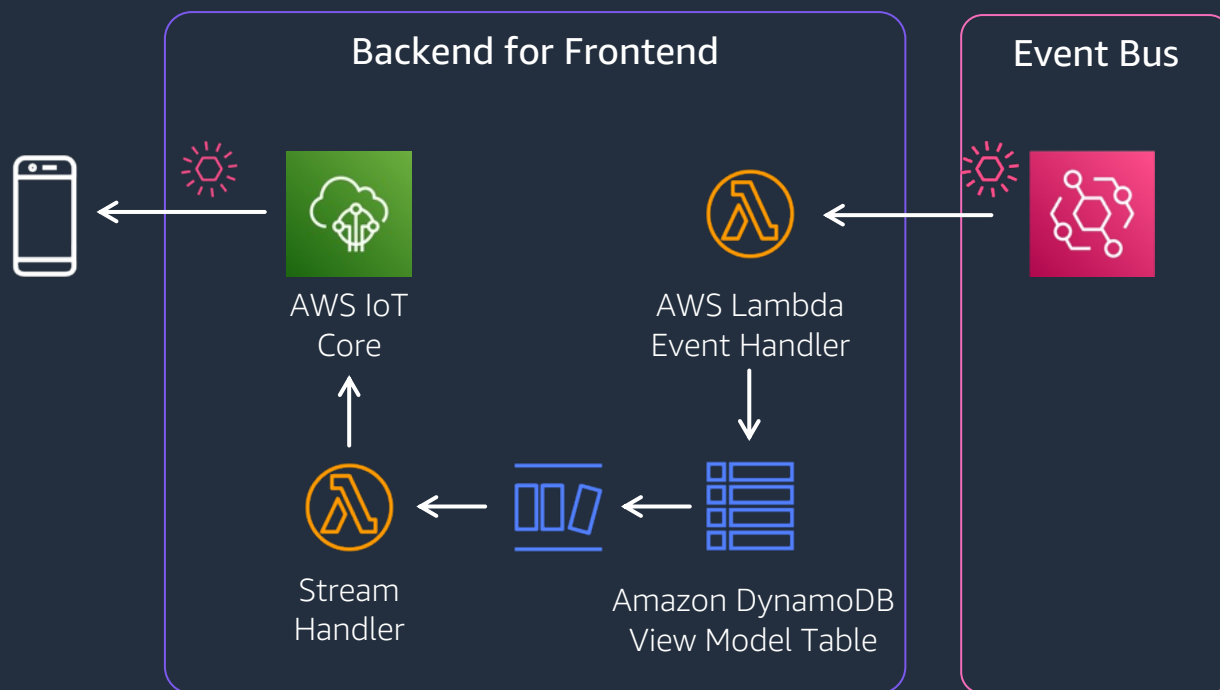
● Public ● IAM/Cognito ● Lambda/Custom ● OIDC

Real-time update w/AWS IoT Core



1. Event handler receives, processes event
2. Store in view model table
3. Data change published to stream (filter as needed)
4. Stream handler **publishes message to IoT core topic**

Real-time update w/AWS IoT Core



- + Highly flexible
- + Broadcast / P2P via topic name
- Single endpoint
- IAM/Cognito or X509
 - Scope IAM permission to topic

Auth Options

● Public ● IAM/Cognito ● Lambda/Custom ● OIDC

Client subscription with AWS IoT Core*

Broadcast subscription

```
await mqttClient.subscribe({
  subscriptions: [
    {
      qos: mqtt5.QoS.AtLeastOnce,
      topicFilter: "orders/events"
    }
  ]
});
```

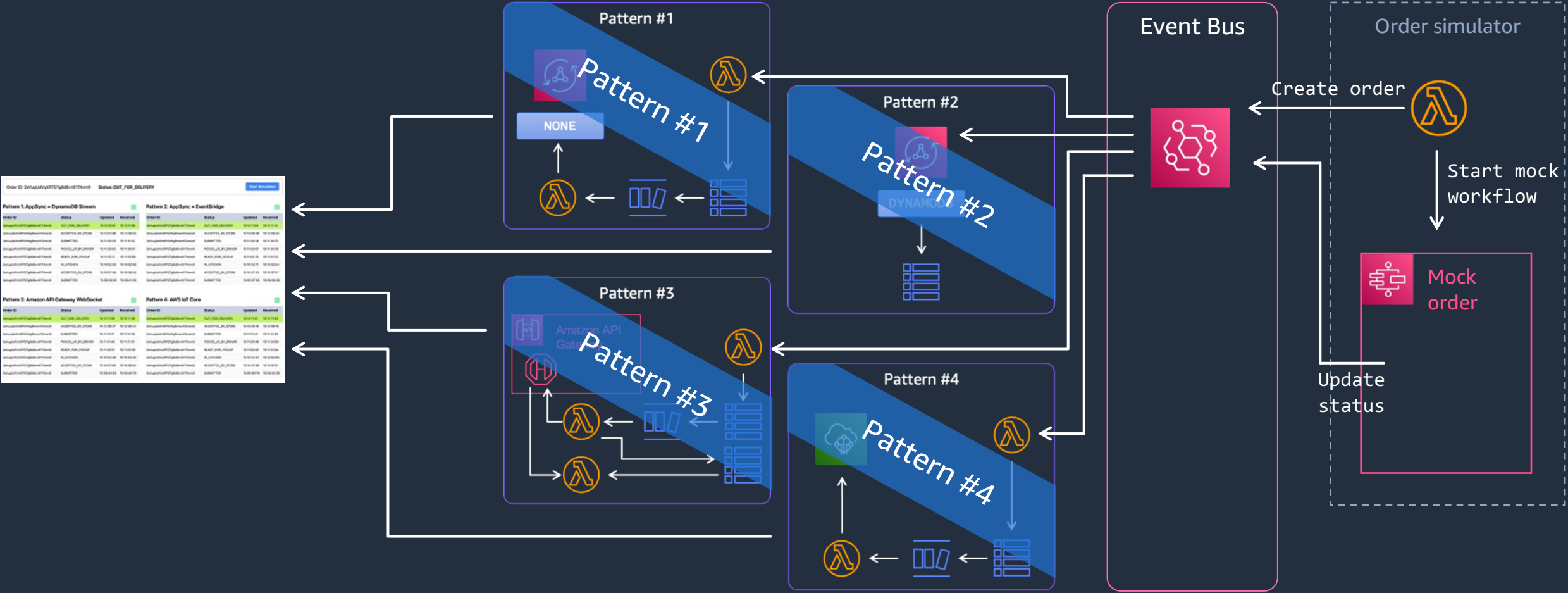
Point-to-point subscription

```
await mqttClient.subscribe({
  subscriptions: [
    {
      qos: mqtt5.QoS.AtLeastOnce,
      topicFilter: "orders/events/ORDER#123"
    }
  ]
});
```

Demo



Demo: Inner-workings



Order ID:

Status:

Start Simulation

Pattern 1: AppSync + DynamoDB Stream



Order ID	Status	Updated	Received
----------	--------	---------	----------

Pattern 2: AppSync + EventBridge



Order ID	Status	Updated	Received
----------	--------	---------	----------

Pattern 3: Amazon API Gateway WebSocket



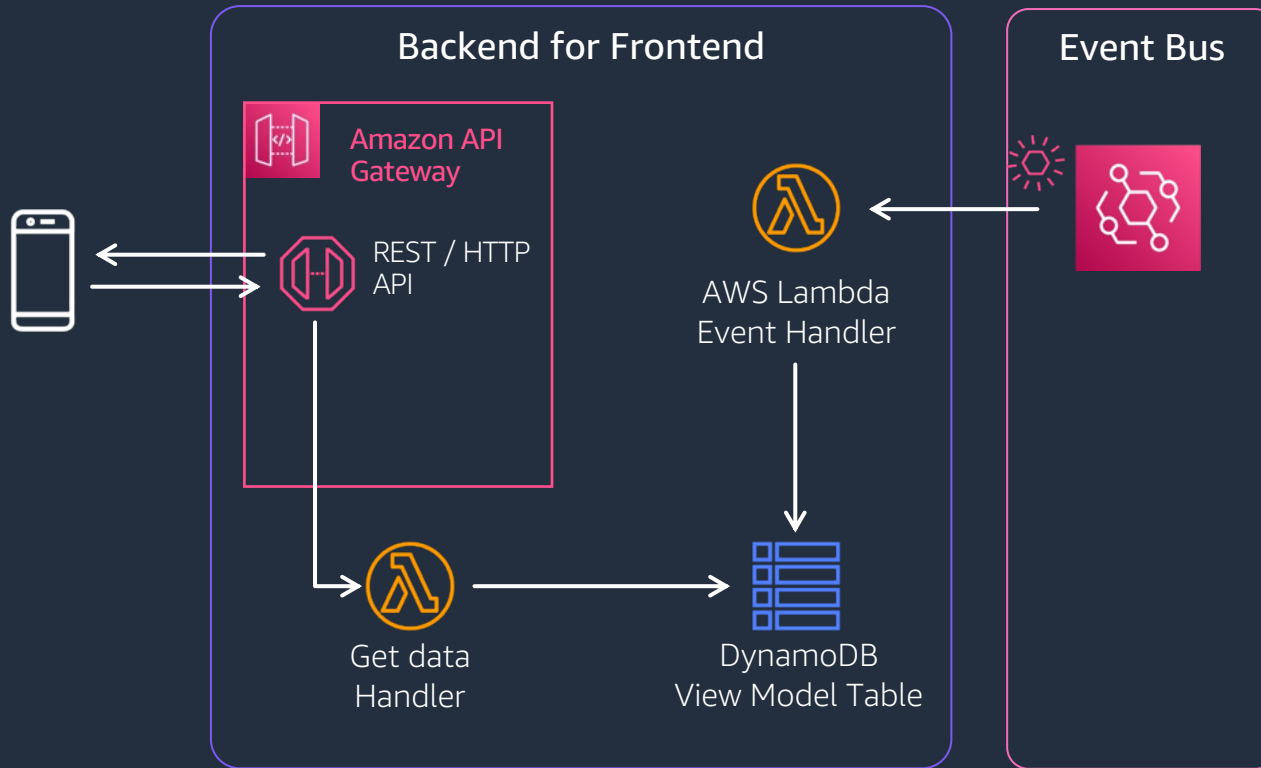
Order ID	Status	Updated	Received
----------	--------	---------	----------

Pattern 4: AWS IoT Core



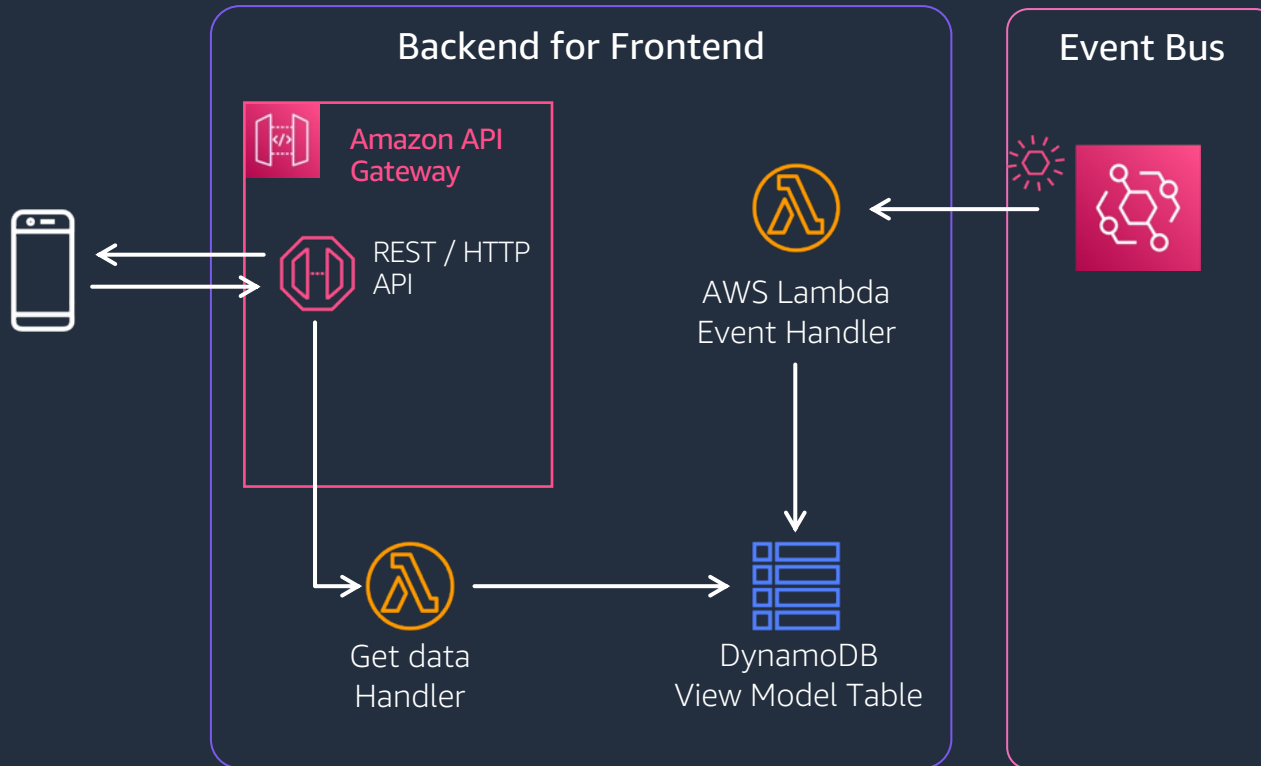
Order ID	Status	Updated	Received
----------	--------	---------	----------

Poll-based update w/Amazon API Gateway



1. Client posts message, receives waiting token
2. Client intermittently polls API for updates

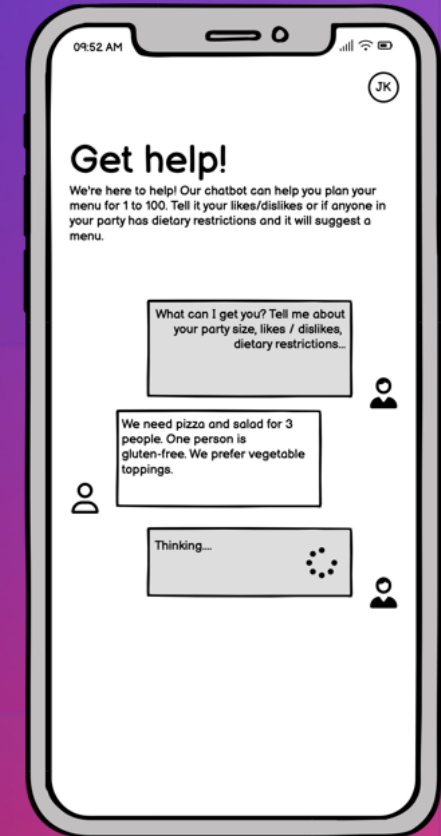
Poll-based update w/Amazon API Gateway



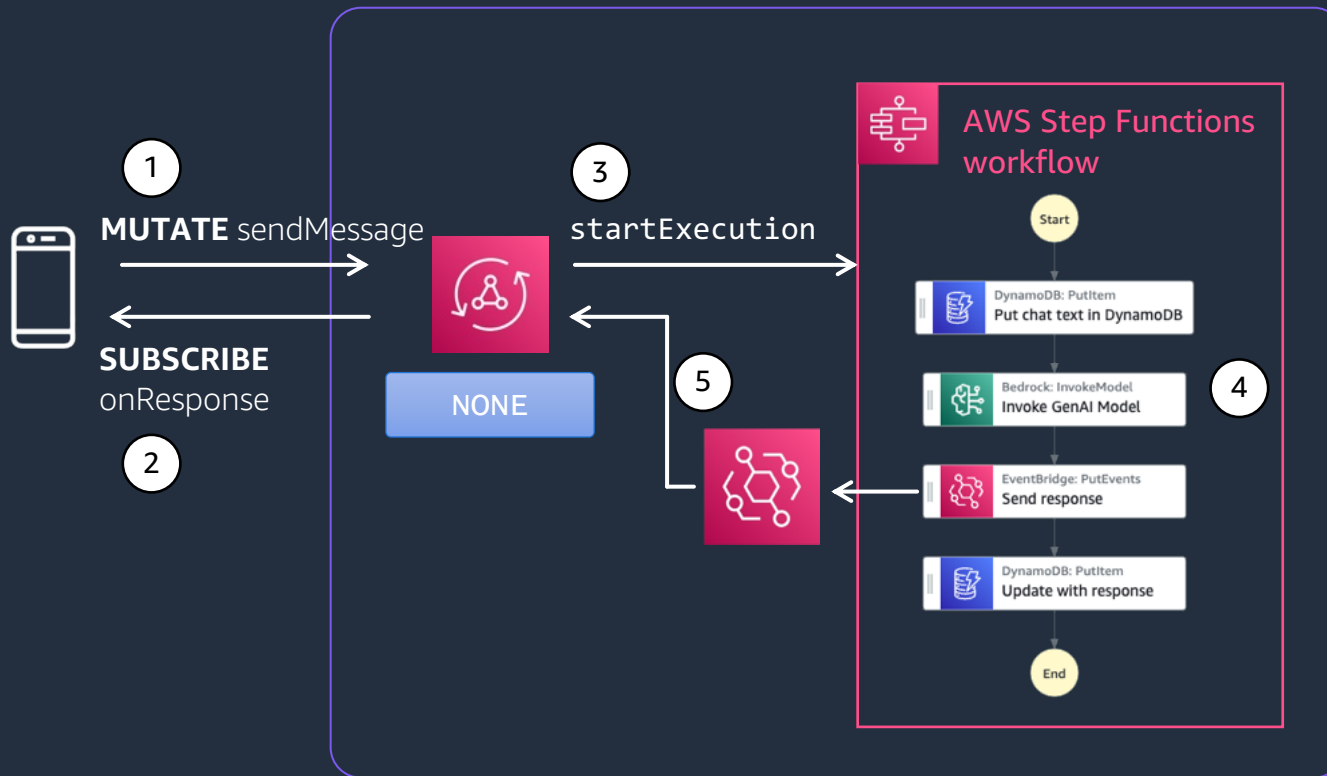
- + Familiar REST semantics
- + Widely supported

- Poll-based, not push
- Intermittent

Long running processes

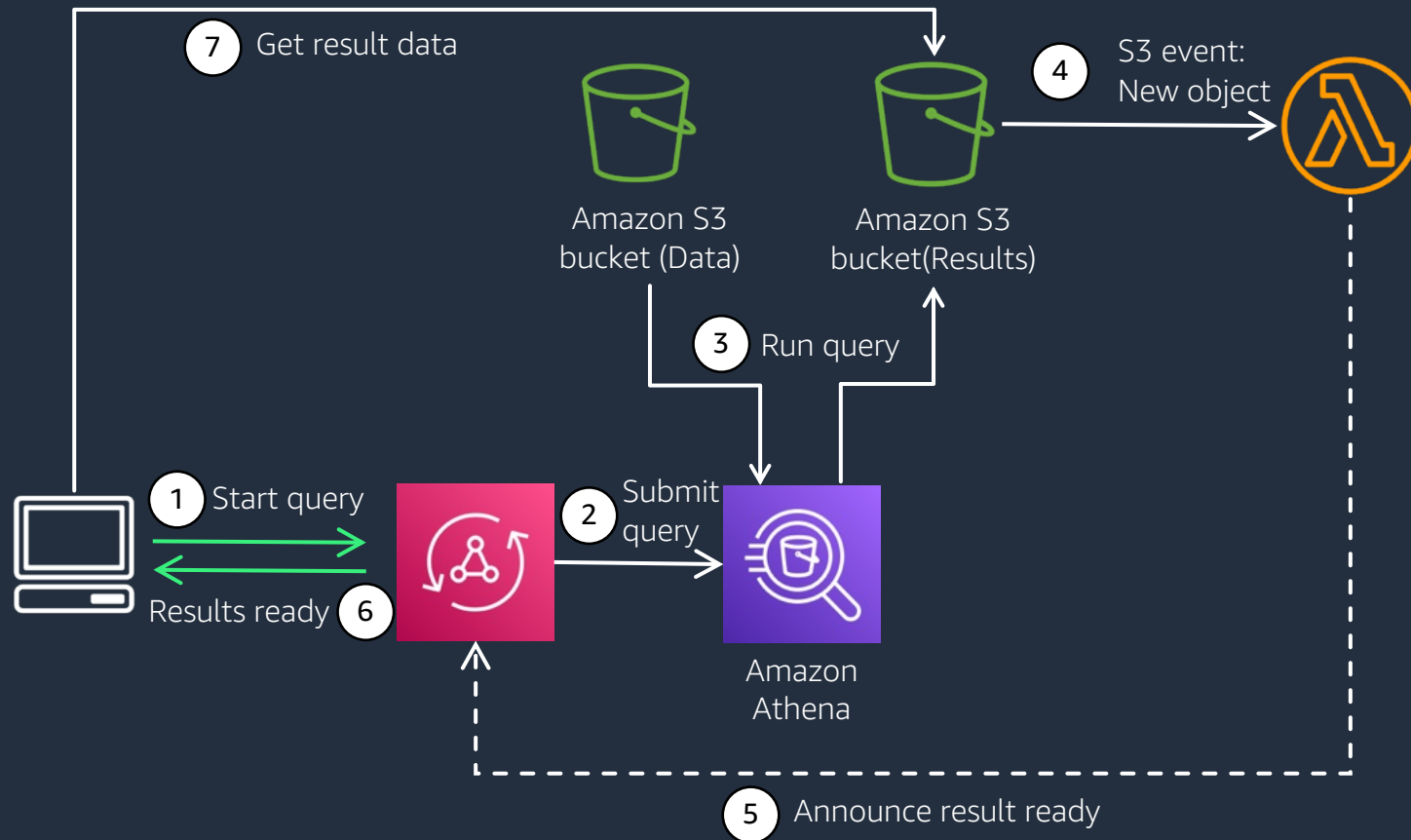


Handle long-running processes



1. Start chat via mutation
 - Return unique chatId
2. Subscribe to **updates** for chatId
3. Start execution of **Step Functions** workflow
4. Invoke **GenAI** model
5. Publish result to NONE data source

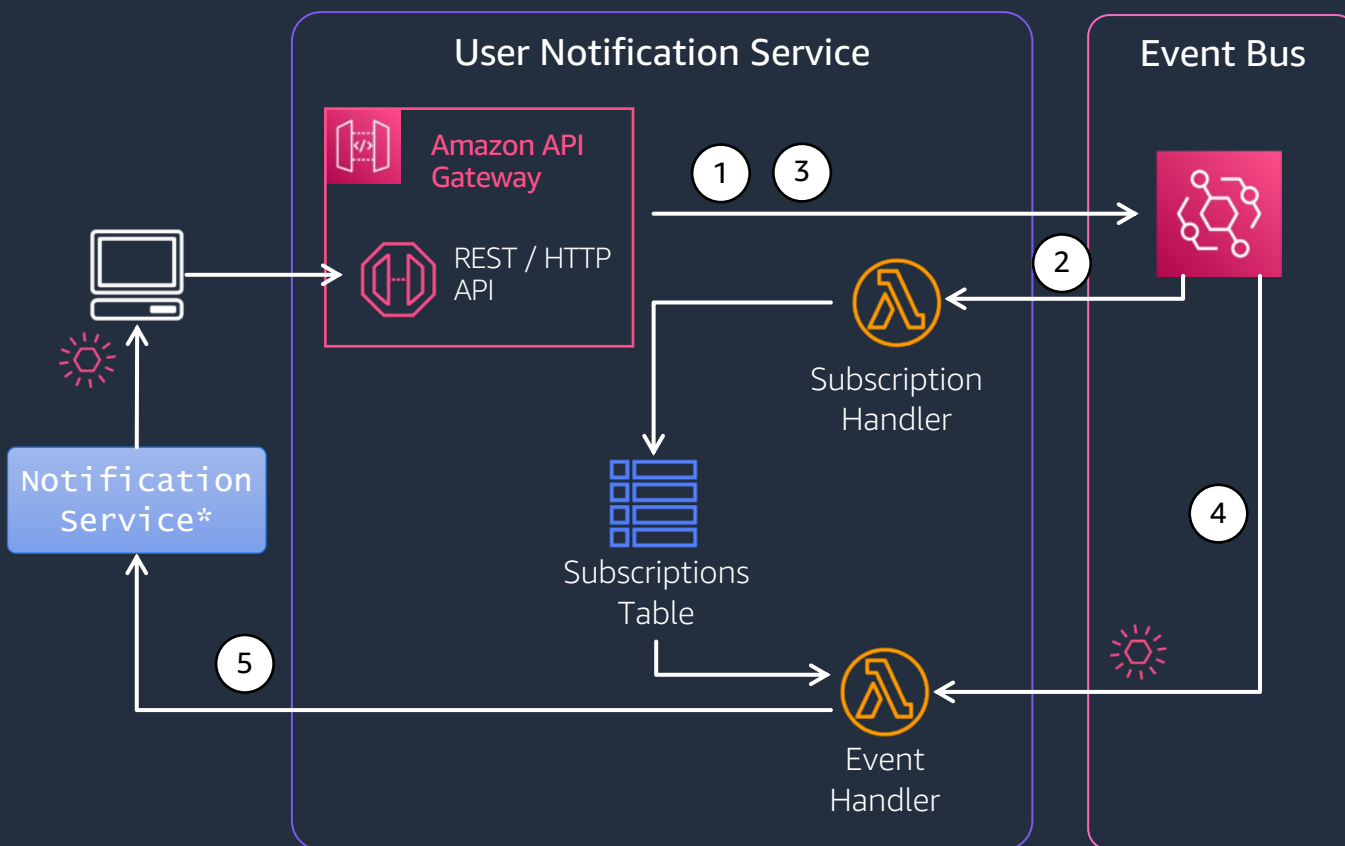
Another use case: Data analysis



Push (notification) events

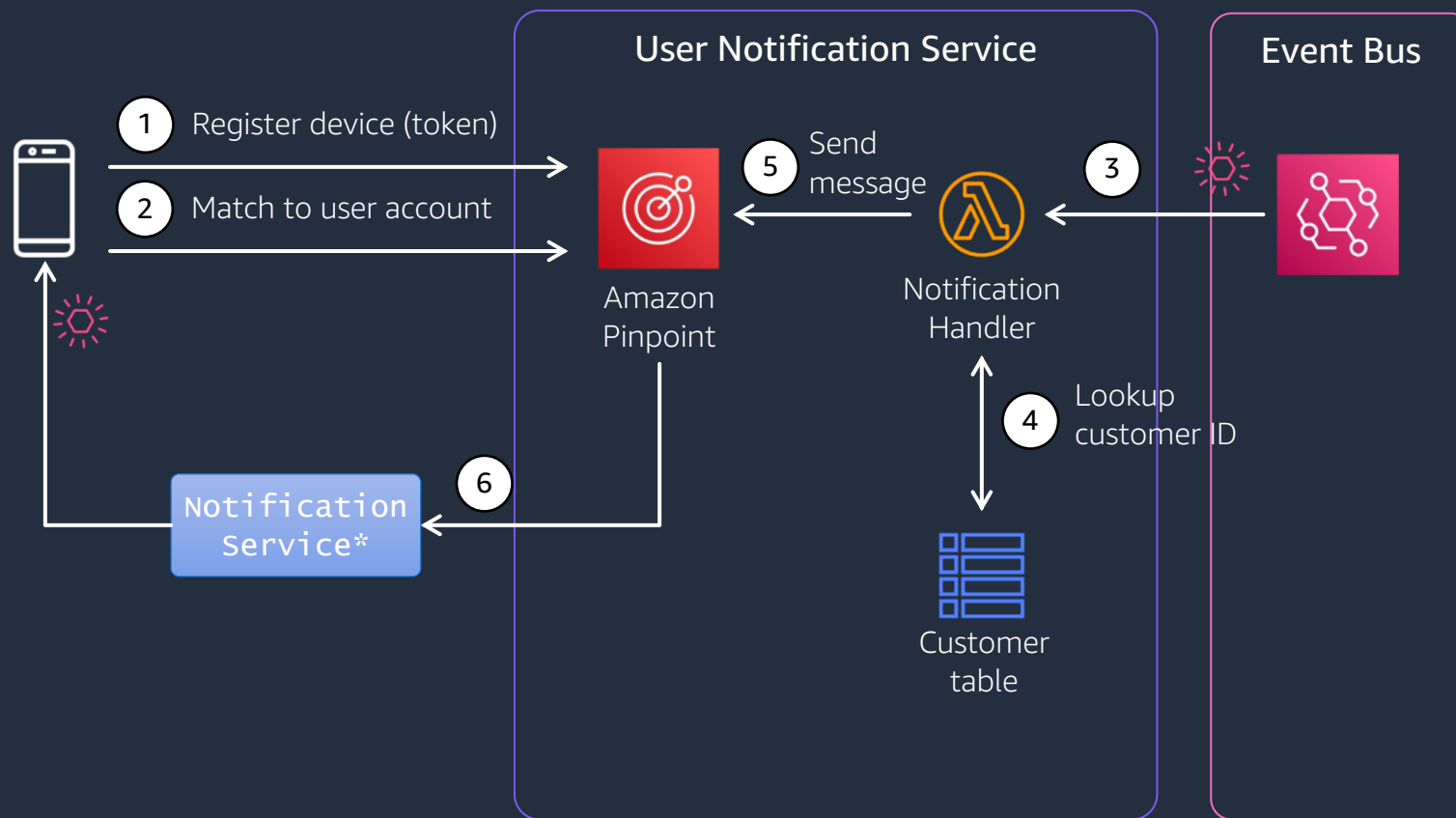


Web push notifications

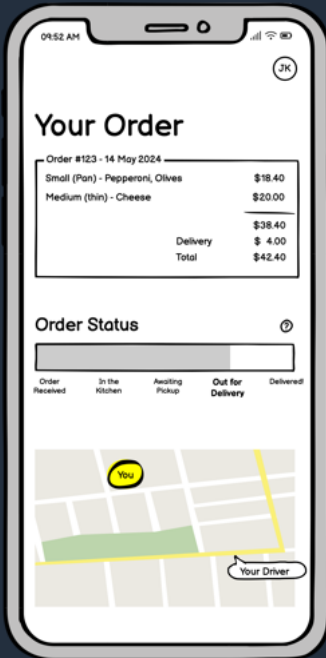


1. **Register** for push notifications
2. Persist **notification endpoint, auth keys**
3. Subscribe to order updates
4. On event, lookup subscribers
5. Publish notification to appropriate **endpoint**

Mobile push notifications

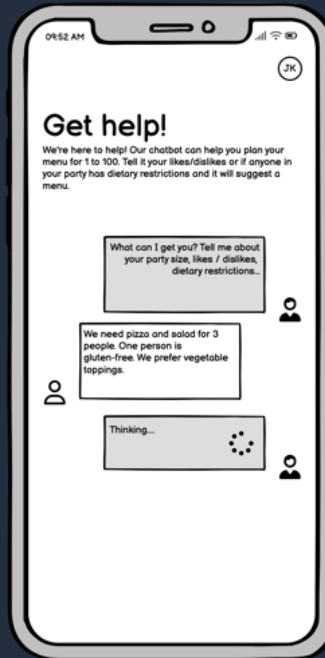


Patterns for event-driven backends



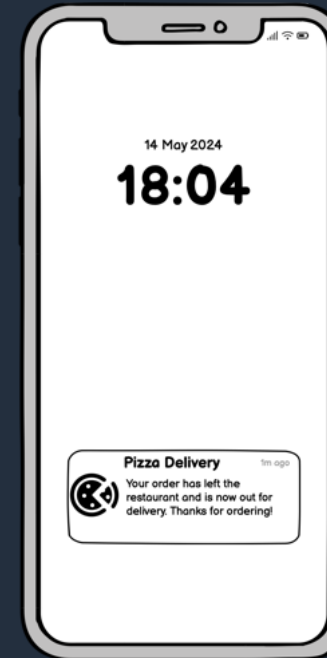
REAL-TIME EVENTS

Something happened, update user interface



LONG RUNNING PROCESS

User is waiting for a long-process (30+ seconds)



PUSH EVENTS

Something happened, alert the user



Find sample code for all patterns at <https://s12d.com/eventdriven-frontend-patterns>

Don't forget to
vote for this session
in the **GOTO Guide app**



Thank you!

Josh Kahn
@joshuaakahn