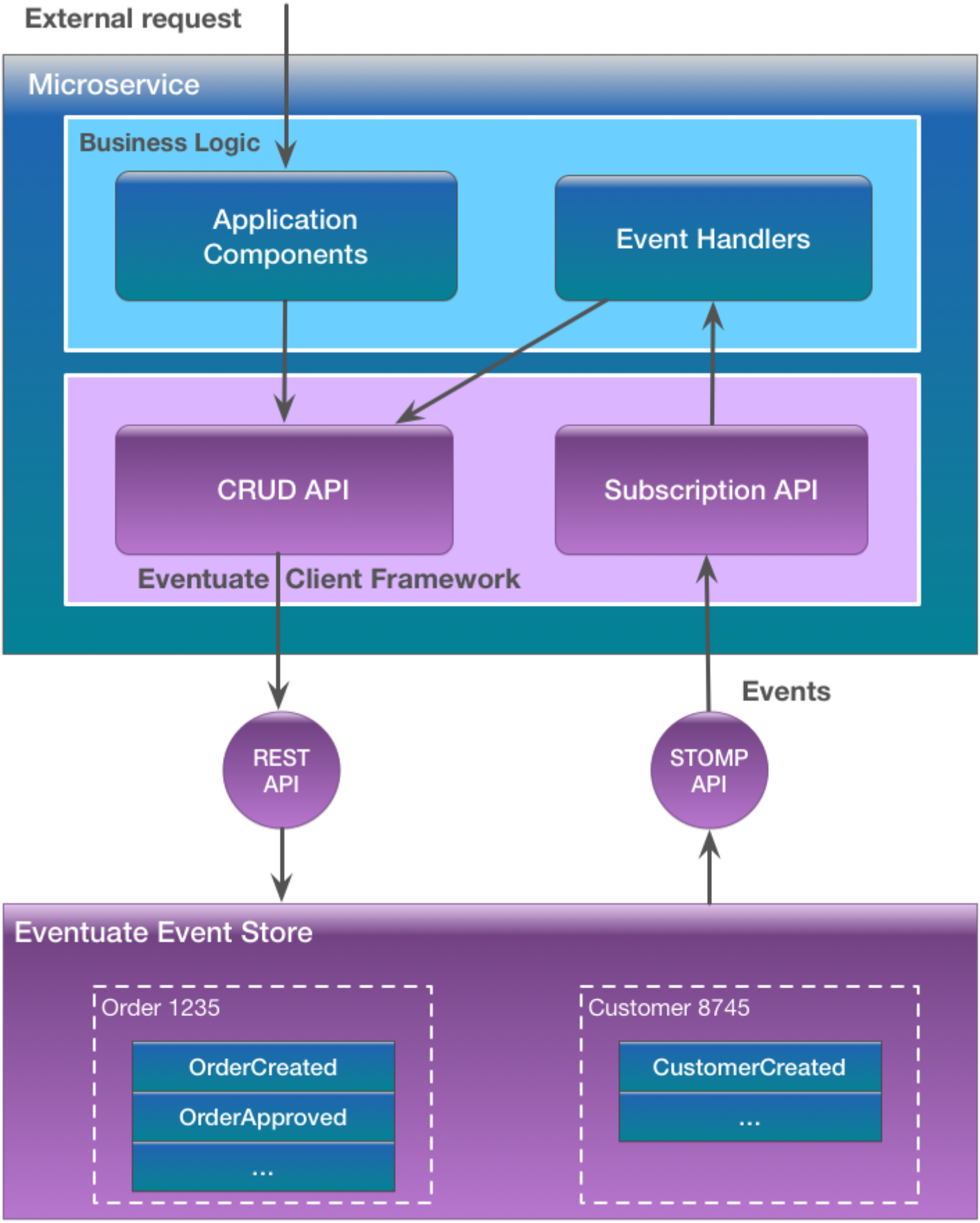
How Eventuate™ works

The Eventuate™ platform consists of a scalable, distributed event store server and client frameworks for various languages and frameworks.

The following diagram shows the key components.



Eventuate Event Store server

The Event Store has two APIs: a REST API and a STOMP API.

The REST API supports the following operations:

- Creating a new entity with an initial, non-empty sequence of events
- Updating an existing entity by appending new events
- Retrieving the events for an entity by primary key

The STOMP API enables microservices to subscribe to events. Microservices use the STOMP API to create durable, named subscriptions to one or more event types. Events that occur when a subscriber is disconnected are delivered when it reconnects.

Eventuate Client Libraries

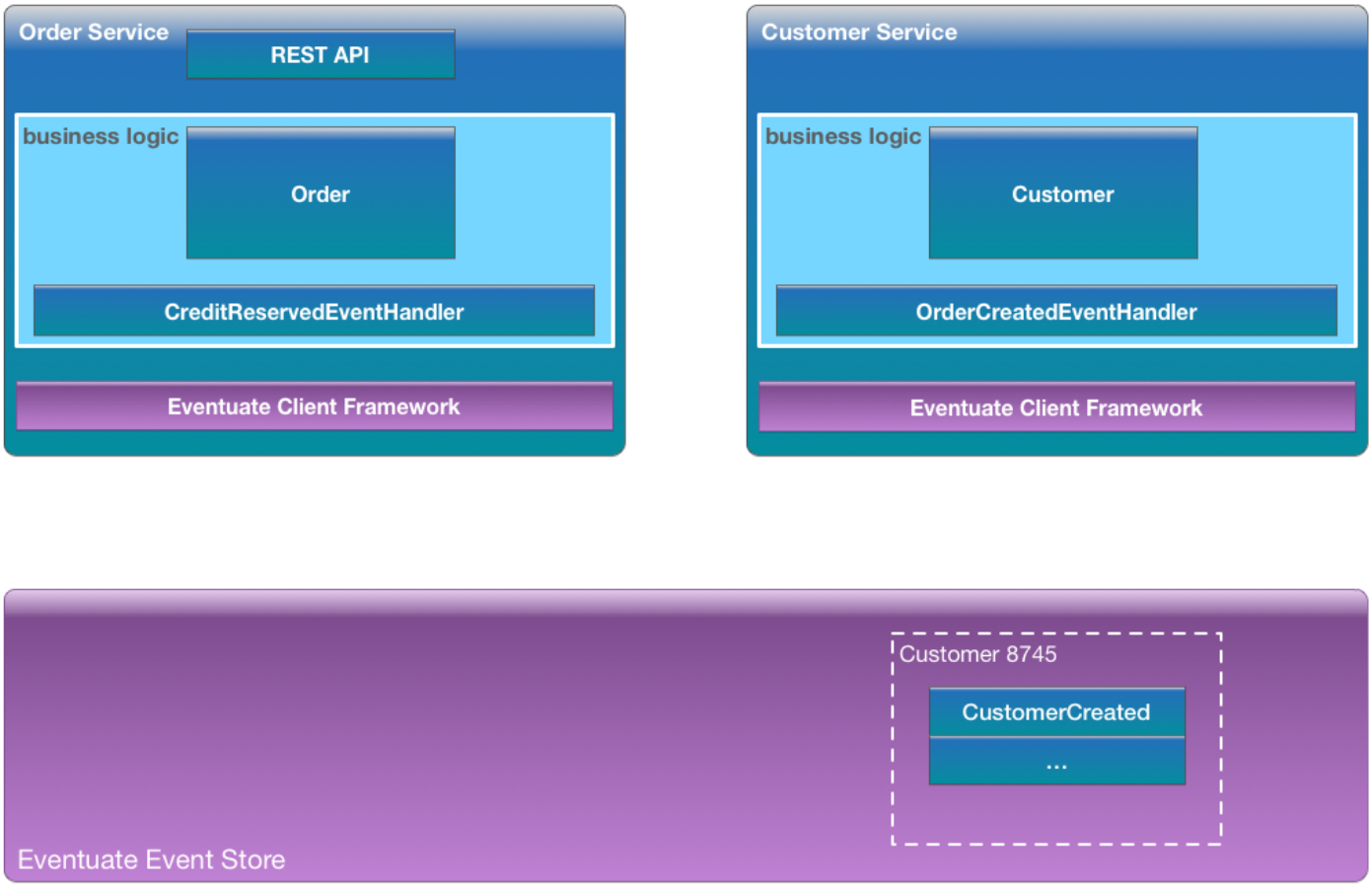
The Eventuate client frameworks build on the server APIs to provide a programming language and framework-specific programming model. For example, Eventuate provides frameworks for Java and Scala developers who use the Spring framework.

You use the client framework to write the following components:

- Event sourcing-based aggregates that process commands and emit domain events
- Services that are invoked by external requests and update aggregates by sending them commands
- Event handlers that subscribe to domain events and do one of three things:
  - Update aggregates
  - Update materialized views
  - Invoke external systems

Implementing eventually consistent transactions

Eventuate makes it easy to implement event-driven, eventually consistent transactions. Click below to how the `Place Order` use case is implemented.



Overview

Each service consists of various application components that are built using the Eventuate client framework.

Order Service consists of the following components:

- Order Service
- Order Aggregate
- Event Handler

Customer Service consists of the following components:

- Customer Aggregate
- Event Handler

A Customer with an ID of 8745 already exists in the event store.

Creating the Order

1. The Order Service receives an HTTP request to create an Order for the Customer.
2. The Service creates an Order aggregate that generates an Order Created event.

3. The client framework persists the Order Created event in the event store.

Handling the Order Created Event

1. The Event store notifies the Customer Service, which has subscribed to Order events.
2. The Eventuate Client framework invokes the event handler for Order Created events.

Retrieving the Customer

1. The OrderCreated event handler loads the Customer from the event store by invoking the loadEntity().
2. The client framework loads the Customer's events from the event store.
3. It reconstructs the Customer by applying the events.

Reserving the Credit

1. The OrderCreated event handler sends the Customer a ReserveCredit command.
2. The Customer performs the credit check and emits a CreditReserved event.
3. The client framework saves the CreditReserved event in the event store.

Handling the Credit Reserved Event

1. The event store notifies the Order Service, which has subscribed to Order events.
2. The client framework invokes the event handler for the Credit Reserved event.

Loading the Order

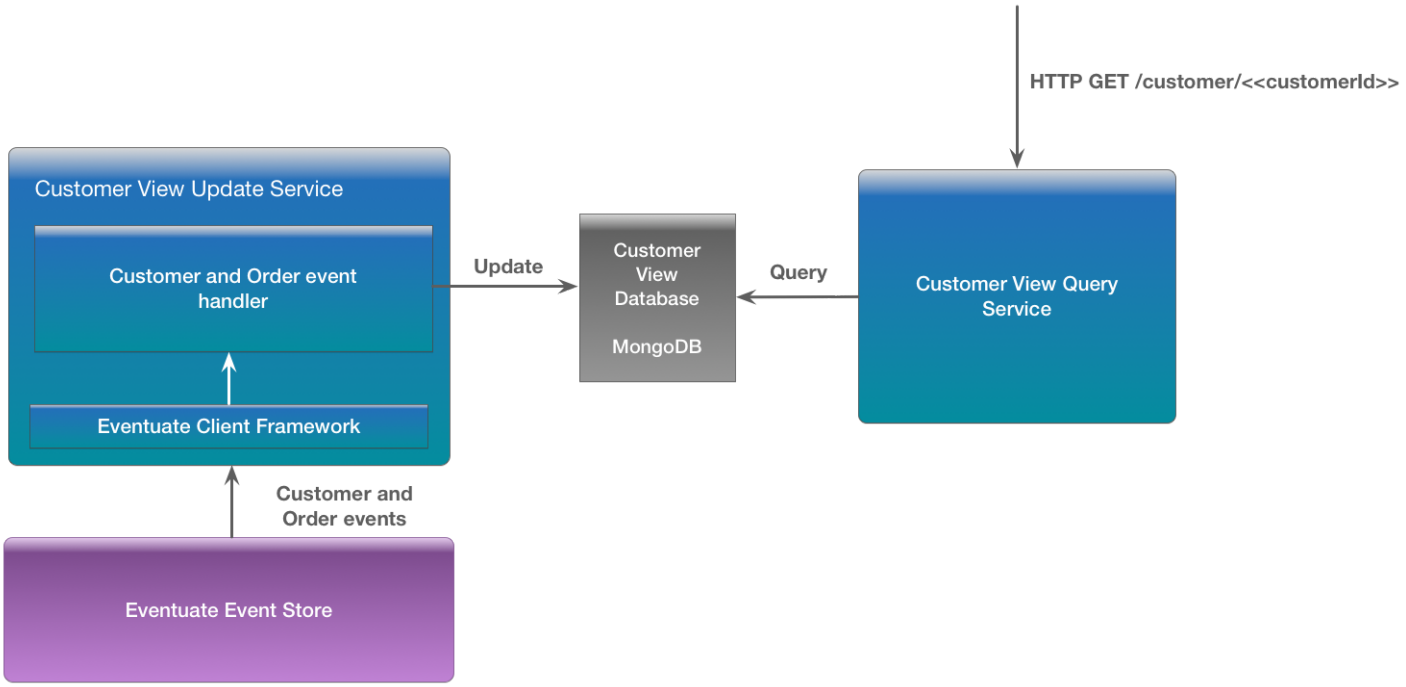
1. The Credit Reserved event handler loads the Order from the event store.

Approving the Order

1. The Credit Reserved event handler notifies the Order that the credit check succeeded.
2. The Order emits an Order Approved event.
3. The client framework saves the Order Approved event in the event store.

Maintaining materialized views

A service can easily maintain a materialized view by subscribing to events published by Eventuate. The following diagram shows how a service can maintain a materialized view of a çustomer and their recent orders using MongoDB.



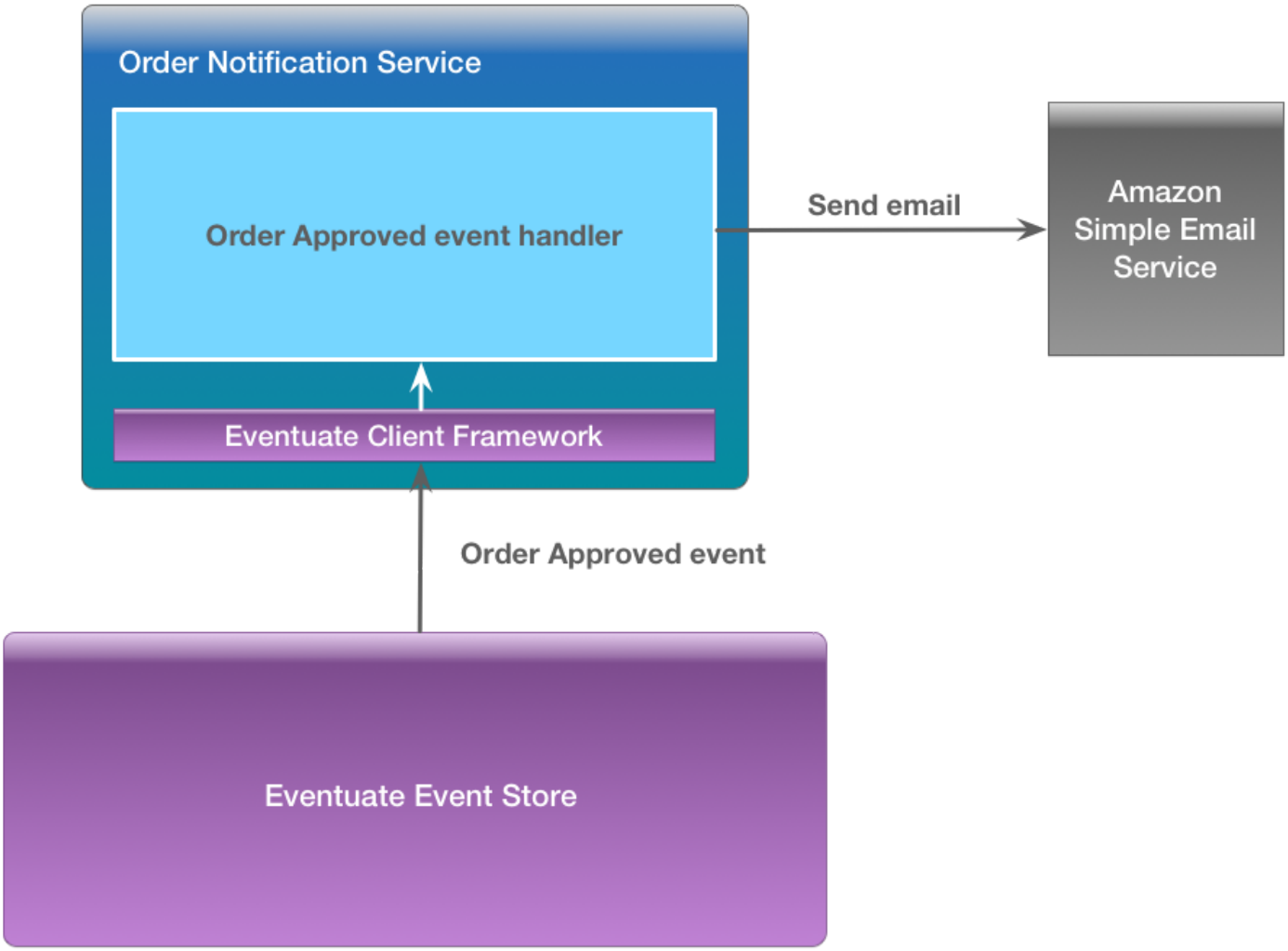
Each customer corresponds to a MongoDB document containing the customer details and recent order.

The `Customer View Update Service` is responsible for updating the materialized view. It defines an event handler that subscribes to the relevant Customer and Order events. When the event handler receives an event, it updates the MongoDB document for that customer.

The `Customer View Query Service` handles a GET request for a customer's details by simply retrieving the MongoDB document for the customer.

Invoking external systems

An application can also register event handlers that invoke external systems. For example, an `Order Notification Service` can register an event handler for `Order Approved` events that send a confirmation email using Amazon SES.



What next

Take a look at the getting started guide and check out the example applications.