Automating Trace Context Propagation with NotificationHandlerOnce

You're absolutely right! We can make the context injection "automagical" by creating a new wrapper struct that implements your NotificationHandlerOnce<T> and IntoNotificationHandlerOnceBox<T> traits. This wrapper will internally manage the crossbeam::channel::Sender<TracedChannelMessage<T>> and handle the OpenTelemetry context injection transparently.
This approach ensures your core T (notification) type remains untouched, while the communication mechanism between your components carries the necessary tracing metadata.

1. Define the TracedChannelMessage Wrapper (Generic)

First, we need a generic wrapper struct that will hold your notification type T and the serialized OpenTelemetry trace context. This struct will be the actual type sent over your crossbeam channels.

```
// src/tracing_channels.rs (or a suitable place for tracing utilities)
use std::collections::HashMap;
use opentelemetry::Context;
use opentelemetry::propagation::{TextMapPropagator, Extractor, Injector};
use opentelemetry::sdk::propagation::TraceContextPropagator;
use tracing::Span;

/// A generic wrapper struct for messages sent over channels that need tracing.
/// It encapsulates the original notification and the OpenTelemetry trace context.
#[derive(Debug, Clone)]
pub struct TracedChannelMessage<T> {
    pub notification: T,
    pub trace_context_carrier: HashMap<String, String>,
}

/// Helper struct for injecting/extracting context into/from a HashMap.
/// This implements the `Injector` and `Extractor` traits required by OpenTelemetry's
`TextMapPropagator`.
pub struct HashMapCarrier<'a>(pub &'a mut HashMap<String, String>);

impl<'a> Injector for HashMapCarrier<'a> {
    fn set(&mut self, key: &str, value: String) {
```

```rust
        self.0.insert(key.to_string(), value);
    }
}

impl<'a> Extractor for HashMapCarrier<'a> {
    fn get(&self, key: &str) -> Option<&str> {
        self.0.get(key).map(|s| s.as_str())
    }

    fn keys(&self) -> Vec<&str> {
        self.0.keys().map(|s| s.as_str()).collect()
    }
}

/// **Injects** the current OpenTelemetry Context into a new HashMap.
/// This function should be called *before* sending a message over a channel,
/// typically within the `handle_notification` implementation of your sender wrapper.
pub fn inject_current_context() -> HashMap<String, String> {
    let propagator = TraceContextPropagator::new();
    // Get the current active context from the tracing span on the *emitter* side.
    let parent_context = Span::current().context();
    let mut carrier = HashMap::new();
    propagator.inject_context(&parent_context, &mut HashMapCarrier(&mut carrier));
    carrier
}

/// **Extracts** an OpenTelemetry Context from a HashMap.
/// This function should be called *after* receiving a message from a channel,
/// on the *receiver* side, before creating new spans.
pub fn extract_context_from_carrier(carrier: &HashMap<String, String>) -> Context {
    let propagator = TraceContextPropagator::new();
    propagator.extract_context(&HashMapCarrier(carrier))
}
```

2. Implement the Wrapper for Your NotificationHandlerOnce Trait

Now, we'll create a new struct, TracedNotificationSender, which will hold your crossbeam::channel::Sender<TracedChannelMessage<T>>. This struct will implement your NotificationHandlerOnce and IntoNotificationHandlerOnceBox traits, performing the context injection when handle_notification is called.

```rust
// src/notification_system.rs (or where your traits are defined)
```

```
use super::tracing_channels::{TracedChannelMessage, inject_current_context}; // Adjust path
as needed
use crossbeam_channel::Sender;
use std::any::type_name;
use tracing::{self, info}; // Import tracing crate for macros and Span

// Your existing traits (no changes needed here)
pub trait NotificationHandlerOnce<T>: Send + Sync {
    fn handle_notification(&self, notification: T);
}

pub trait IntoNotificationHandlerOnceBox<T> {
    fn into_notification_handler_once_box(self) -> Box<dyn NotificationHandlerOnce<T>>;
}

// --- Your new implementation for TracedChannelMessage ---

/// A wrapper around a `crossbeam::channel::Sender` that automatically injects trace context.
/// This struct implements your `NotificationHandlerOnce` trait.
pub struct TracedNotificationSender<T> {
    sender: Sender<TracedChannelMessage<T>>,
}

impl<T> TracedNotificationSender<T> {
    /// Creates a new `TracedNotificationSender` from an underlying
`crossbeam_channel::Sender`.
    pub fn new(sender: Sender<TracedChannelMessage<T>>) -> Self {
        Self { sender }
    }
}

// Implement NotificationHandlerOnce for our TracedNotificationSender.
// This is where the "automagic" context injection happens.
impl<T> NotificationHandlerOnce<T> for TracedNotificationSender<T>
where
    T: Send + Sync, // Your notification type must be Send + Sync
{
    fn handle_notification(&self, notification: T) {
        // --- Automagical Context Injection ---
        // When `handle_notification` is called (i.e., a notification is emitted),
        // we automatically capture the OpenTelemetry context from the *current* active tracing
span.
        let trace_context_carrier = inject_current_context();
```

```
        // Wrap the original notification with the captured trace context.
        let traced_message = TracedChannelMessage {
            notification, // Your original notification data
            trace_context_carrier, // The context carrier
        };

        // Send the wrapped message through the internal crossbeam channel.
        if let Err(err) = self.sender.send(traced_message) {
            tracing::warn!("Failed to send traced notification {}: {:?}", type_name::<T>(), err);
        } else {
            info!("Successfully sent traced notification {}", type_name::<T>());
        }
    }
}

// Implement IntoNotificationHandlerOnceBox for TracedNotificationSender.
// This allows you to convert your `TracedNotificationSender` into a trait object.
impl<T> IntoNotificationHandlerOnceBox<T> for TracedNotificationSender<T>
where
    T: Send + Sync + 'static, // Your notification type must be Send + Sync and 'static
{
    fn into_notification_handler_once_box(self) -> Box<dyn NotificationHandlerOnce<T>> {
        Box::new(self)
    }
}

// --- Your existing implementation for Sender (Consider if you still need this) ---
// If you want all notifications to be traced, you might remove or deprecate this.
// If you need untraced paths, keep it and explicitly choose which sender to use.
/*
impl<T> NotificationHandlerOnce<T> for Sender<T>
where
    T: Send + Sync,
{
    fn handle_notification(&self, notification: T) {
        if let Err(err) = self.send(notification) {
            tracing::warn!("Failed to send {}: {:?}", type_name::<T>(), err);
        }
    }
}

impl<T> IntoNotificationHandlerOnceBox<T> for Sender<T>
```

```
where
   T: Send + Sync + 'static,
{
   fn into_notification_handler_once_box(self) -> Box<dyn NotificationHandlerOnce<T>> {
      Box::new(self)
   }
}
*/
```

3. How to Use It in Your Application

3.1. Setting up the Channels

When you create your crossbeam::channels, they will now be typed for TracedChannelMessage<T>. You will then wrap the Sender with your TracedNotificationSender.

```
// In your main application setup (e.g., src/main.rs)
// Make sure to import the necessary modules:
use crossbeam_channel::{unbounded, Receiver, Sender};
use crate::tracing_channels::{TracedChannelMessage, extract_context_from_carrier}; // Your
tracing utilities
use crate::notification_system::{NotificationHandlerOnce, IntoNotificationHandlerOnceBox,
TracedNotificationSender}; // Your notification traits and sender wrapper
use tracing::{info, span, Level};
use opentelemetry::Context; // Important for Context::attach()

// Define a sample notification type (your original event struct)
#[derive(Debug, Clone)]
pub struct MyNotification {
   pub user_id: String,
   pub client_order_id: String,
   pub data: String,
}

// ... (OpenTelemetry and tracing subscriber initialization as in previous examples) ...

fn setup_notification_pipeline() {
   // Create the crossbeam channel for traced messages
   let (tx_raw, rx_raw): (Sender<TracedChannelMessage<MyNotification>>,
Receiver<TracedChannelMessage<MyNotification>>) = unbounded();

   // Wrap the raw sender with our TracedNotificationSender
```

```
    let traced_sender = TracedNotificationSender::new(tx_raw);

    // Now, convert it into a boxed trait object for distribution to emitters
    let boxed_handler: Box<dyn NotificationHandlerOnce<MyNotification>> =
traced_sender.into_notification_handler_once_box();

    // Pass `boxed_handler` to the components that will emit notifications.
    // E.g., `my_async_websocket_server.set_notification_handler(boxed_handler.clone());`
    // Or `my_processor_thread.set_output_handler(boxed_handler.clone());`

    // Spawn a non-async thread to consume notifications
    std::thread::spawn(move || {
        process_notifications(rx_raw); // Pass the raw receiver to the consumer
    });
}

// This function represents a component that emits notifications
#[tracing::instrument]
fn emit_notification(handler: &dyn NotificationHandlerOnce<MyNotification>, user_id: String,
client_order_id: String, data: String) {
    info!("Emitting notification for UserID: {}, ClientOrderID: {}", user_id, client_order_id);
    // Add correlation IDs to the current span for better visibility in Kibana
    tracing::Span::current().record("user.id", &user_id);
    tracing::Span::current().record("client.order_id", &client_order_id);

    let notification = MyNotification {
        user_id,
        client_order_id,
        data,
    };
    handler.handle_notification(notification); // This call will automagically inject the context!
}
```

3.2. Receiving and Attaching Context in Worker Threads

On the receiving end of the channel, your worker threads will consume TracedChannelMessage<T>. They will then need to explicitly extract the context and attach it to their current thread before creating any new spans.

```
// In your worker thread that consumes from the channel (e.g., src/worker.rs)

pub fn process_notifications(receiver: Receiver<TracedChannelMessage<MyNotification>>) {
    info!("Notification processor thread started.");
    for traced_message in receiver {
```

```
        // --- Automagical Context Extraction and Attachment ---
        // 1. Extract the OpenTelemetry Context from the received message's carrier.
        let parent_otel_context =
extract_context_from_carrier(&traced_message.trace_context_carrier);

        // 2. Attach this context to the current thread.
        // This makes the extracted context the "active" context for this thread.
        // Any new `tracing` spans created within this block will automatically become
        // children of the span represented by `parent_otel_context`.
        let _guard = parent_otel_context.attach();

        // 3. Create a new tracing span for the processing of this specific notification.
        // This span will automatically link to the parent span from where the notification was
sent.
        let processing_span = span!(
            Level::INFO,
            "handle_incoming_notification",
            notification_type = std::any::type_name::<MyNotification>(),
            user_id = &traced_message.notification.user_id, // Add correlation IDs as span
attributes
            client_order_id = &traced_message.notification.client_order_id
        );
        let _processing_guard = processing_span.enter();

        // --- Your original notification handling logic goes here ---
        info!("Handling notification: {:?}", traced_message.notification);

        // Simulate some work
        std::thread::sleep(std::time::Duration::from_millis(20));

        // If this processing triggers further notifications/sends (e.g., to another handler),
        // the *current* span's context (`handle_incoming_notification`) will automatically be
captured
        // when `handle_notification` is called again on another `TracedNotificationSender`.

        // The `_processing_guard` drops here, ending the `handle_incoming_notification` span.
        // The `_guard` drops here, detaching the parent context from the thread.
    }
    info!("Notification processor thread stopped.");
}
```

4. Full Example Structure

To make this runnable, your project structure might look like this:

```
your_project/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── notification_system.rs  // Contains your traits and TracedNotificationSender
│   ├── tracing_channels.rs     // Contains TracedChannelMessage and context helpers
│   └── worker.rs               // Example of a notification consumer
```

And your main.rs would orchestrate the setup:

```
// src/main.rs
use opentelemetry::{
    global,
    sdk::{
        trace::{self, Sampler},
        Resource,
    },
    KeyValue,
};
use opentelemetry_otlp::{self, WithExportConfig};
use tracing::{info, Level};
use tracing_subscriber::{prelude::*, registry::Registry, EnvFilter};
use crossbeam_channel::{unbounded, Sender};

mod notification_system;
mod tracing_channels;
mod worker; // Assuming worker.rs contains the process_notifications function

use notification_system::{MyNotification, NotificationHandlerOnce,
IntoNotificationHandlerOnceBox, TracedNotificationSender};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // 1. Configure OpenTelemetry Tracer Provider
    let otlp_exporter = opentelemetry_otlp::new_exporter()
        .http()
        .with_endpoint("http://localhost:4318/v1/traces") // OTLP Collector endpoint
        .with_timeout(std::time::Duration::from_secs(3))
        .build()?;
```

```
    let tracer = opentelemetry_otlp::new_pipeline()
        .tracing()
        .with_exporter(otlp_exporter)
        .with_trace_config(
            trace::config()
                .with_sampler(Sampler::AlwaysOn)
                .with_resource(Resource::new(vec![
                    KeyValue::new("service.name", "event-driven-rust-app"),
                    KeyValue::new("service.version", "0.1.0"),
                ])),
        )
        .install_batch(opentelemetry::runtime::Tokio)?;

    // 2. Create tracing-opentelemetry layer
    let opentelemetry_layer = tracing_opentelemetry::OpenTelemetryLayer::new(tracer);

    // 3. Create tracing subscriber for console output and OpenTelemetry export
    let subscriber = Registry::default()
        .with(EnvFilter::from_default_env())
        .with(opentelemetry_layer)
        .with(tracing_subscriber::fmt::layer().compact());

    global::set_subscriber(subscriber)?;

    info!("Application starting...");

    // Setup the notification pipeline
    let (tx_raw, rx_raw) = unbounded();
    let traced_sender = TracedNotificationSender::new(tx_raw);
    let boxed_handler: Box<dyn NotificationHandlerOnce<MyNotification>> =
traced_sender.into_notification_handler_once_box();

    // Spawn the notification consumer thread
    std::thread::spawn(move || {
        worker::process_notifications(rx_raw);
    });

    // Simulate emitting some notifications from an async context
    // (e.g., your Axum WebSocket server would do this)
    tokio::spawn(async move {
        let handler_clone = boxed_handler.clone();
        for i in 0..3 {
            let user_id = format!("user_{}", i);
```

```
        let client_order_id = format!("order_{}", i);
        let data = format!("data_payload_{}", i);

        // Use an instrumented block to simulate the context from the async server
        let server_span = tracing::span!(Level::INFO, "websocket_server_request", user.id =
&user_id, client.order_id = &client_order_id);
        let _guard = server_span.enter();

        info!("Simulating server request for UserID: {}", user_id);
        notification_system::emit_notification(handler_clone.as_ref(), user_id, client_order_id,
data);
        tokio::time::sleep(std::time::Duration::from_millis(100)).await;
      }
  });

  // Keep main alive for a bit to allow tasks to run
  tokio::time::sleep(std::time::Duration::from_secs(5)).await;

  info!("Application finished.");
  global::shutdown_tracer_provider(); // Ensure all traces are flushed
  Ok(())
}
```

5. Verification

Run your OpenTelemetry Collector (as described in previous responses) and then your Rust application. You should observe:

1. **Continuous Traces**: In Kibana APM, you'll see single, unbroken traces starting from the websocket_server_request span, flowing through the handle_incoming_notification span in the worker thread, and continuing if that worker emits further traced notifications.
2. **Correlation IDs**: The user.id and client.order_id attributes will be present on all relevant spans within the trace, allowing you to easily filter and search for all operations related to a specific user or order.

This solution effectively leverages Rust's trait system and OpenTelemetry's context propagation mechanisms to provide comprehensive tracing for your complex, hybrid async/non-async event-driven architecture, without requiring modifications to your core business event types.