

Speedrun Client with the Matrix Rust SDK

Prelude

Once upon a time...

Swedish for *Matrix*



```
$ cargo new --bin matris
$ cd matris
$ cargo add \
    matrix-sdk \
    matrix-sdk-ui \
    matrix-sdk-sqlite \
    + friends
$ hx src/main.rs
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    logger();

    let options = argh::from_env();
    let client = client(&options).await?;
    let client = session(client, &options).await?;

    app(client).await?;

    Ok(())
}
```

Client

- The almighty client
- Entry point or central point to all data and API
- Keeps things isolated/contained inside the Client
- Comes with a lot of configurations
 - Use the builder!

```
async fn client(options: &bin::Options) -> Result<Client, Error> {
    let bin::Options { server_name, session_path } = options;

    let client_builder = Client::builder()
        .store_config(
            StoreConfig::new(CrossProcessLockConfig::MultiProcess {
                holder_name: "matrix".to_owned(),
            })
            .crypto_store(CryptoStore::open(/* ... */).await?)
            .state_store(StateStore::open(/* ... */).await?)
            .event_cache_store(EventCacheStore::open(/* ... */).await?),
        )
        .server_name_or_homeserver_url(&server_name)
        .with_encryption_settings(EncryptionSettings {
            auto_enable_cross_signing: true,
            backup_download_strategy: BackupDownloadStrategy::AfterDecryptionFailure,
            auto_enable_backups: true,
        });

    Ok(client_builder.build().await?)
}
```

Session

- The client needs to be logged in to send or receive data
- Two session mechanisms
 1. Matrix Auth (legacy)
 2. OAuth 2.0 (recommended)
- A session can be restored

```
async fn session(client: Client, options: &bin::Options) -> Result<Client, Error> {  
    // ...  
  
    if let Ok(maybe_session) = fs::read_to_string(&session_path) {  
        let session: MatrixSession = serde_json::from_str(&maybe_session)?;  
        client.restore_session(session).await?;  
    }
```

```
} else {  
    println!("Logging in with username and password...");  
  
    loop {  
        let username = /* ... */;  
        let password = /* ... */;  
  
        match client.matrix_auth().login_username(&username, password).await {  
            Ok(_) => {  
                println!("Logged in as {username}");  
                break;  
            }  
            Err(error) => {  
                println!("Error logging in: {error}");  
                println!("Please try again\n");  
            }  
        }  
    }  
}
```

```
// Immediately save the session to disk.
if let Some(session) = client.session() {
    let AuthSession::Matrix(session) = session else { unreachable!() };
    fs::write(session_path, serde_json::to_string(&session)?)?;

    println!("Session saved");
}
}

Ok(client)
}
```

Demonstrative CLI application

1. **Model** contains the data
 - **Model::render()** draws the data on the screen
 - **Model::update()** applies a **Message** onto the model
2. **Message** represents all possible actions
 - Terminal inputs are mapped to **Message**
3. Let's write a modal client, because... it's fun
 - (This is not about the SDK, just for demo purposes)

```
// input.rs
```

```
pub enum Input {
    Redraw,
    KeyPress(KeyEvent),
    RoomListUpdate(Vec<VectorDiff<Room>>),
    TimelineUpdate(Vec<VectorDiff<Arc<TimelineItem>>>),
}

pub async fn map_input_to_message(
    input_receiver: &mut Receiver<Input>,
    model: &app::Model,
) -> Option<app::Message> {
    match input_receiver.recv().await? {
        Input::Redraw => None,

        Input::KeyPress(key_event) => map_key_event_to_message(key_event, model),

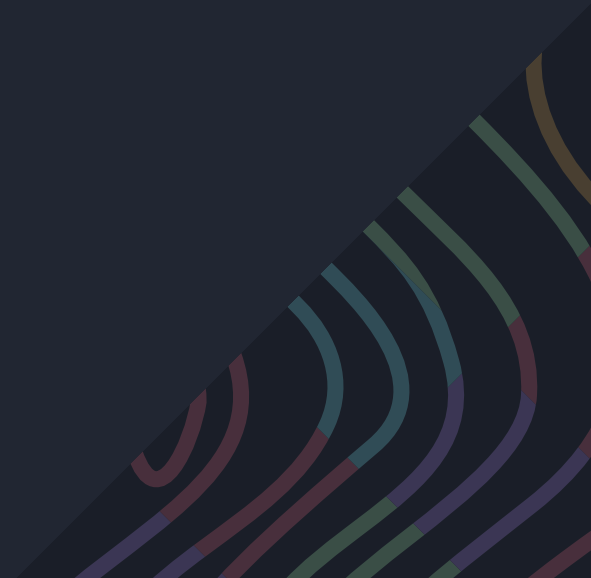
        Input::RoomListUpdate(diffs) => {
            Some(app::Message::RoomList(mode::room_list::Message::UpdateRoomList(diffs)))
        }

        Input::TimelineUpdate(diffs) => Some(app::Message::Room(room::Message::Timeline(
            timeline::Message::UpdateTimeline(diffs),
        ))),
    }
}
```

```
// input.rs
```

```
fn map_key_event_to_message(key_event: KeyEvent, model: &app::Model) -> Option<app::Message> {  
    Some(match key_event.code {  
        KeyCode::Esc => app::Message::SetMode(app::Mode::None),  
        code => match &model.mode {  
            app::Mode::None => match code {  
                KeyCode::Char('q') => app::Message::Quit,  
                // ...  
            },  
  
            app::Mode::Space(_) => app::Message::Space(match code {  
                KeyCode::Char('f') => mode::space::Message::OpenRoomList,  
                KeyCode::Char('S') => mode::space::Message::StartSyncService,  
                KeyCode::Char('s') => mode::space::Message::StopSyncService,  
                KeyCode::Char('c') => mode::space::Message::EmptyEventCache,  
                KeyCode::Char('l') => mode::space::Message::OpenLogger,  
                _ => return None,  
            }),  
  
            // ...  
        })  
    }  
}
```

okay



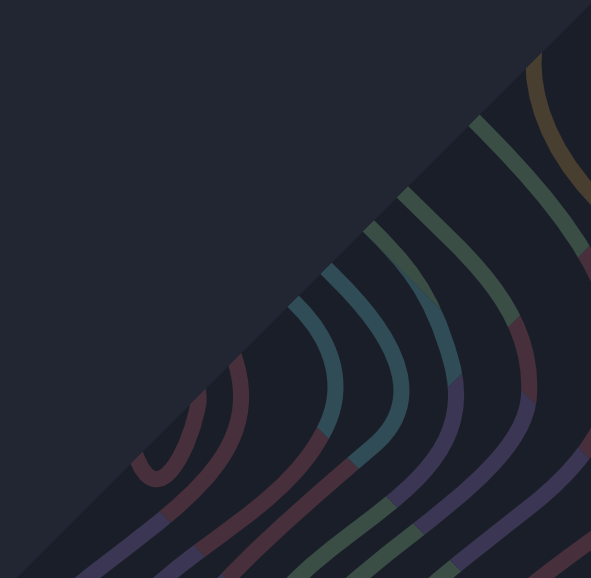
```
// main.rs
```

```
async fn app(client: Client) -> Result<(), Error> {  
    let mut terminal = /* ... */  
  
    let app_result = app::App::new(client).await?.run(&mut terminal).await;  
  
    // ...  
  
    app_result  
}
```

```
// app.rs
```

```
impl App {  
    // ...  
  
    pub async fn run(mut self, terminal: &mut DefaultTerminal) -> Result<(), Error> {  
        // ...  
  
        // Run the app.  
        while !self.model.exit {  
            // Render the app.  
            terminal.draw(|frame| self.model.render(...));  
  
            // Handle inputs and get a `Message` in return.  
            let mut next_message =  
                input::map_input_to_message(&mut self.input_receiver, &self.model).await;  
  
            // Process the `Message` and the subsequent `Message`s if any are  
            // chained.  
            while let Some(message) = next_message {  
                next_message = self.model.update(message).await;  
            }  
        }  
  
        // ...  
    }  
}
```

okay



Sync service

- Runs 2 sync loops for free, automagically managed by the Matrix Rust SDK!
 - Sync new events (Sliding Sync behind the scene)
 - Sync encryption (E2EE made easy!)
- 2 methods:
 - **start** and **stop**
- Comes with some configurations
 - e.g. offline mode support
 - Use the builder!

```
// app.rs
```

```
pub struct Model {  
    pub sync_service: SyncService,  
    // ...  
}
```

```
impl Model {  
    pub async fn new(client: Client, /* ... */) -> Result<Self, Error> {  
        let sync_service =  
            SyncService::builder(client).with_offline_mode().build().await?;  
  
        sync_service.start().await;  
  
        Ok(Self {  
            sync_service,  
            // ...  
        })  
    }  
  
    // ...  
}
```

```
impl App {  
    // ...  
  
    pub async fn run(...) -> ... {  
        // ...  
  
        } // while !self.model.exit  
  
        self.model.sync_service.stop().await;  
  
        Ok(())  
    }
```

Room List service

- To list, filter, sort all the rooms
 - `SyncService::room_list_service()`
 - Get the room items and a controller with `RoomList::entries_with_dynamic_adapters()`
 - Precise the page size
 - Add one page, or reset to one page
 - Set the filter
 - This will also start the stream of updates
- Filters:
- `all`
 - `any`
 - `category`
 - `deduplicate_versions`
 - `favourite`
 - `fuzzy_match_room_name`
 - `identifiers`
 - `invite`
 - `joined`
 - `low_priority`
 - `non_left`
 - `none`
 - `normalized_match_room_name`
 - `not`
 - `read_receipts`
 - `space`

Sorters:

- `latest_event`
- `lexicographic`
- `name`
- `recency`

```
// mode/room_list.rs
```

```
pub struct Model {  
    rooms: Vector<Room>,  
    // ...  
}
```

```
impl Model {  
    pub async fn new(sync_service: Arc<SyncService>, input_sender: Sender<Input>) -> Self {  
        let room_list_service = sync_service.room_list_service();  
  
        let (room_list_controller_sender, room_list_controller_receiver) = oneshot::channel();  
  
        let _room_list_updates_handle = spawn(room_list_updates_task(  
            room_list_service,  
            room_list_controller_sender,  
            input_sender,  
        ))  
        .abort_on_drop();  
  
        let room_list_controller = room_list_controller_receiver.await.unwrap();  
  
        room_list_controller.set_filter(new_filter_non_left().into());  
  
        // ...  
    }  
}
```

```
async fn room_list_updates_task(
    room_list_service: Arc<RoomListService>,
    room_list_controller_sender: Sender<RoomListDynamicEntriesController>,
    input_sender: Sender<Input>,
) {
    // Get a `RoomList`.
    let room_list = room_list_service.all_rooms().await.unwrap();

    // Subscribe to updates + get a controller.
    let (rooms_stream, room_list_controller) =
        room_list.entries_with_dynamic_adapters(u16::MAX as usize);

    let _ = room_list_controller_sender.send(room_list_controller);

    // Broadcast updates.
    let mut rooms_stream = pin!(rooms_stream);

    while let Some(diffs) = rooms_stream.next().await {
        let _ = input_sender.send(Input::RoomListUpdate(diffs)).await;
    }
}
```

```
pub enum Message {  
    UpdateRoomList(Vec<VectorDiff<Room>>),  
    UpdateFilter(KeyEvent),  
    Select,  
    // ...  
}
```

```
// eyeball-im crate
```

```
pub enum VectorDiff<T> {  
    Append { values: Vector<T> },  
    Clear,  
    PushFront { value: T },  
    PushBack { value: T },  
    PopFront,  
    PopBack,  
    Insert { index: usize, value: T },  
    Set { index: usize, value: T },  
    Remove { index: usize },  
    Truncate { length: usize },  
    Reset { values: Vector<T> },  
}
```

```
impl Model {  
    pub async fn update(&mut self, message: Message) -> Option<app::Message> {  
        match message {  
            Message::UpdateRoomList(diffs /* : Vec<VectorDiff<Room>> */) => {  
                for diff in diffs {  
                    diff.apply(&mut self.rooms);  
                }  
  
                // ...  
            }  
        }  
    }  
}
```

```
Message::UpdateFilter(key_event) => {  
    let search_term = /* ... */;  
  
    // Reset the filter.  
    if search_term.is_empty() {  
        self.room_list_controller  
            .set_filter(new_filter_non_left().into());  
    }  
    // Search for a room by its name!  
    else {  
        self.room_list_controller.set_filter(new_filter_all(vec![  
            new_filter_non_left().into(),  
            new_filter_fuzzy_match_room_name(&search_term).into(),  
        ]).into());  
    }  
  
    // ...  
}
```

```
Message::Select => {  
    let index = /* ... */;  
    let (room, _) = self.rooms.get(index)?;  
  
    app::Message::OpenRoom(room.clone())  
}
```

Room

- Provides the type (joined, invited etc.), avatar, display name, members, parent space, notification mode, and so!

Some API:

- `cached_display_name`
- `is_direct`
- `is_call`
- `is_public`
- `is_space`
- `num_unread_mentions`
- `num_unread_messages`
- `num_unread_notifications`
- `topic`
- `avatar`
- `encryption_state`
- `members`
- ...

```
// app.rs
```

```
impl Model {  
    pub async fn update(&mut self, message: Message) -> Option<Message> {  
        match message {  
            Message::OpenRoom(room) => {  
                self.sync_service  
                    .room_list_service()  
                    .set_room_subscriptions(&[room.room_id()])  
                    .await;  
  
                self.room = Some(room::Model::new(room).await);  
  
                // ...  
            }  
            // ...  
        }  
    }  
}
```

Timeline

- Provides a simple way to display events/messages, based on **VectorDiff**, again
- It enhances some Room API, like a simple way to send, edit or redact events/messages

Some API:

- `send`
- `send_gallery`
- `send_location`
- `send_reply`
- `send_attachment`
- `retry_send`
- `abort_send`
- `toggle_reaction`
- `edit`
- `paginate_backwards`
- `paginate_forwards`
- `subscribe`

```
// timeline.rs
```

```
pub struct Model {  
    timeline: Arc<Timeline>,  
    items: Vector<Arc<TimelineItem>>,  
    // ...  
}
```

```
impl Model {  
    pub async fn new(room: &Room) -> Self {  
        let timeline = Arc::new(room.timeline_builder().build().await.unwrap());  
        let items = Vector::new();  
  
        let items_updates_task_handle = spawn(items_updates_task(timeline.clone(), /* ... */))  
            .abort_on_drop();  
  
        // ...  
    }  
}
```

```
async fn items_updates_task(timeline: Arc<Timeline>, input_sender: Sender<Input>) {  
    let (initial_items, items_stream) = timeline.subscribe().await;  
  
    // Initial items.  
    let _ = input_sender  
        .send(Input::TimelineUpdate(vec![VectorDiff::Reset { values: initial_items }]))  
        .await;  
  
    // Broadcast updates.  
    let mut items_stream = pin!(items_stream);  
  
    while let Some(diffs) = items_stream.next().await {  
        let _ = input_sender.send(Input::TimelineUpdate(diffs)).await;  
    }  
}
```

```
impl Model {  
    pub async fn update(&mut self, message: Message) -> Option<app::Message> {  
        match message {  
            Message::UpdateTimeline(diffs) => {  
                for diff in diffs {  
                    diff.apply(&mut self.items);  
                }  
            }  
        }  
  
        // ...  
    }  
}
```

```
Message::PaginateBackwards => {  
    let _ = self.timeline.paginate_backwards(20).await;  
}  
  
// ...
```

```
Message::ToggleReactionOnLastMessage => {  
    if let Some(last_timeline_item_id) =  
        self.items.iter().rev().find_map(|timeline_item| {  
            let event_timeline_item = timeline_item.as_event()?;  
            event_timeline_item.content().as_message()?;  
  
            Some(event_timeline_item.identifier())  
        })  
    {  
        self.timeline.toggle_reaction(&last_timeline_item_id, "👍").await.unwrap();  
    }  
}  
  
// ...
```

```
pub fn render(&self, /* ... */) {  
    let items = self.items  
        .iter()  
        .rev()  
        .filter_map(|item| self.render_timeline_item(item, /* ... */));  
  
    // ...  
}
```

```
pub fn render_timeline_item<'a>(
    &self,
    item: &'a Arc<TimelineItem>,
    /* ... */
) -> Option<ListItem<'a>> {
    Some(ListItem::new(match item.kind() {
        TimelineItemKind::Virtual(virtual_item) => {
            // ...
        }

        TimelineItemKind::Event(event_item) => {
            // ...
        },
    )))
}
```

```
TimelineItemKind::Virtual(virtual_item) => match virtual_item {
  VirtualTimelineItem::DateDivider(time) => {
    let time = if let Some(time) = time.to_system_time() {
      DateTime::<Local>::from(time).format("%a, %e %b %Y").to_string()
    } else {
      "date divider".to_owned()
    };

    let text = format!("—— {time} ——");

    // ...
  }
  VirtualTimelineItem::ReadMarker => return None,
  VirtualTimelineItem::TimelineStart => {
    let text = format!("—— Beginning of the room ——");

    // ...
  }
},

// ...
```

```
TimelineItemKind::Event(event_item) => {  
    let sender = /* ... */;  
    let time = /* ... */;  
    let message = /* ... */;  
    let is_local_echo = /* ... */;  
    let reactions = /* ... */;  
    let read_receipts = /* ... */;  
    let own_message = /* ... */;  
}
```

```
let sender = if let TimelineDetails::Ready(Profile {
    display_name: Some(display_name),
    ..
}) = event_item.sender_profile()
{
    display_name.as_str()
} else {
    event_item.sender().as_str()
};
```

```
let time = if let Some(time) = event_item.timestamp().to_system_time() {  
    DateTime::::from(time).format("%H:%M").to_string()  
} else {  
    "???".to_owned()  
};
```

```
let message = match event_item.content() {
  TimelineItemContent::MsgLike(message_like) => match &message_like.kind {
    MsgLikeKind::Message(message) => message.body(),
    MsgLikeKind::UnableToDecrypt(_) => { /* ... */ }
    MsgLikeKind::Redacted => { /* ... */ }
    MsgLikeKind::Poll(_) => { /* ... */ }
    MsgLikeKind::Sticker(_) => { /* ... */ }
    MsgLikeKind::LiveLocation(_) => { /* ... */ }
    MsgLikeKind::Other(_) => { /* ... */ }
  },
  TimelineItemContent::MembershipChange(_) => { /* ... */ }
  TimelineItemContent::ProfileChange(_) => { /* ... */ }
  TimelineItemContent::OtherState(_) => { /* ... */ }
  TimelineItemContent::CallInvite => { /* ... */ }
  TimelineItemContent::RtcNotification { .. } => { /* ... */ }
  TimelineItemContent::FailedToParseMessageLike { .. } => { /* ... */ }
  TimelineItemContent::FailedToParseState { .. } => { /* ... */ }
};
```

```
let is_local_echo = event_item.is_local_echo();
```

```
let reactions = event_item.reactions().iter().map(|(reaction, senders)| {  
    let number_of_senders = senders.len();  
  
    if number_of_senders > 1 {  
        format!("{}", reaction, number_of_senders)  
    } else {  
        format!("{}", reaction)  
    }  
});
```

```
let read_receipts = event_item  
  .read_receipts()  
  .iter()  
  .map(|(user_id, _receipt)| user_id.localpart());
```

```
let own_message = event_item.is_own();
```

Send Queue

- The Sync Service receives events; the Send Queue sends events
- Automatic retries on errors
- Automatic retries on network connectivity
- Seamlessly used by the Timeline

Event Cache

- Reactive database of events
- Starts when the Client is built and activated
- Efficiently organises events in *Caches* from sync and paginations
 - Room, Thread, Pinned-Events, Event-focused (permalink)
- API that listens to the Event Cache:
 - Timeline
 - Latest Events
 - Search
- Seamlessly handles red decryption with **Redecryptor**

Thanks!

- github.com/matrix-org/matrix-rust-sdk
 - My work on the Matrix Rust SDK is being sponsored by [Element](#)
- codeberg.org/hywan/matrix
- Some technical articles about the Matrix Rust SDK
 - [From 19k to 4.2M events/sec: story of a SQLite query optimisation \(mnt.io\)](#)
 - [About memory pressure, lock contention, and Data-oriented Design \(mnt.io\)](#)
 - [Switching higher-order streams to first-order streams \(mnt.io\)](#)
- [mnt.io](#), [@hywan](#) on Mastodon