Prerequisites

1. Install required toolchains and libs

```
https://tauri.app/v1/guides/getting-started/prerequisites
```

(Or scan the QR code)

2. Install the wasm32 Rust target

```
rustup target install wasm32-unknown-unknown
```

3. Install the Tauri CLI

```
cargo install tauri-cli
```

4. Install Trunk

```
cargo install trunk
```

5. Check out the Code

```
git clone https://github.com/crabnebula-dev/rustnation23-workshop
```

Hands-on with Tauri & Yew

Intro Me

CrabNebula

ToC

Prerequisites

1. Install the wasm32 Rust target

```
rustup install wasm32-unknown-unknown
```

2. Install the Tauri CLI

```
cargo install tauri-cli
```

3. Install Trunk

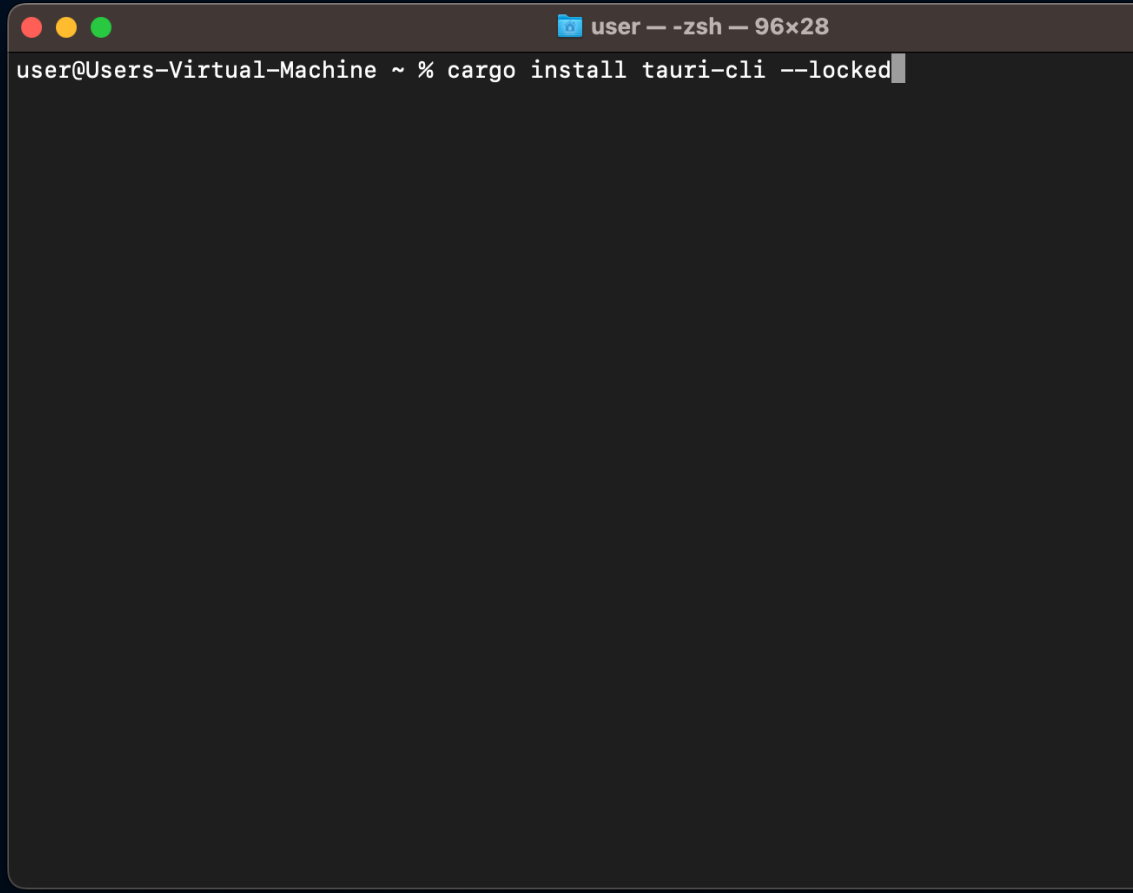
```
cargo install trunk
```

4. Check out the Code

```
git clone https://github.com/crabnebula-dev/rustnation23-workshop
```

Tauri CLI

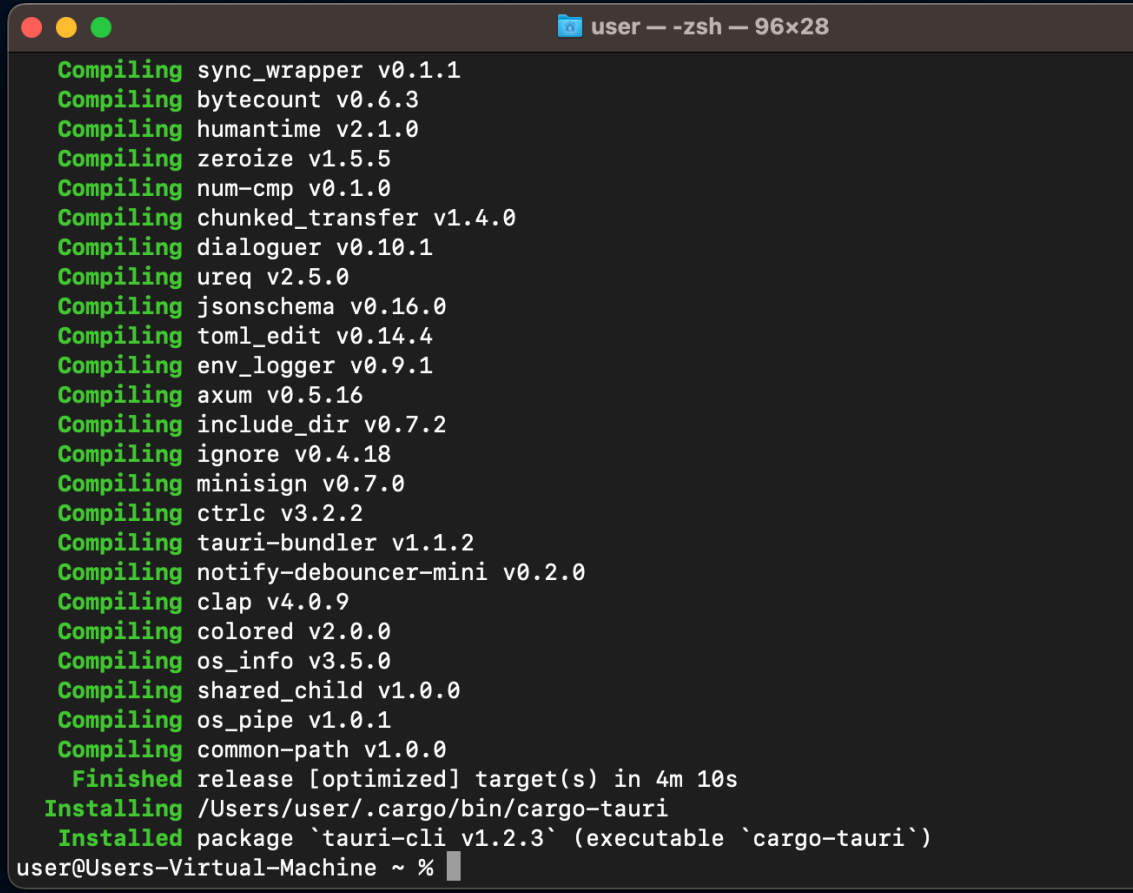
- Development Runner
- Build-tool and Packager
- Generate Icons
- Generate Signing Keys



CrabNebula

Tauri CLI

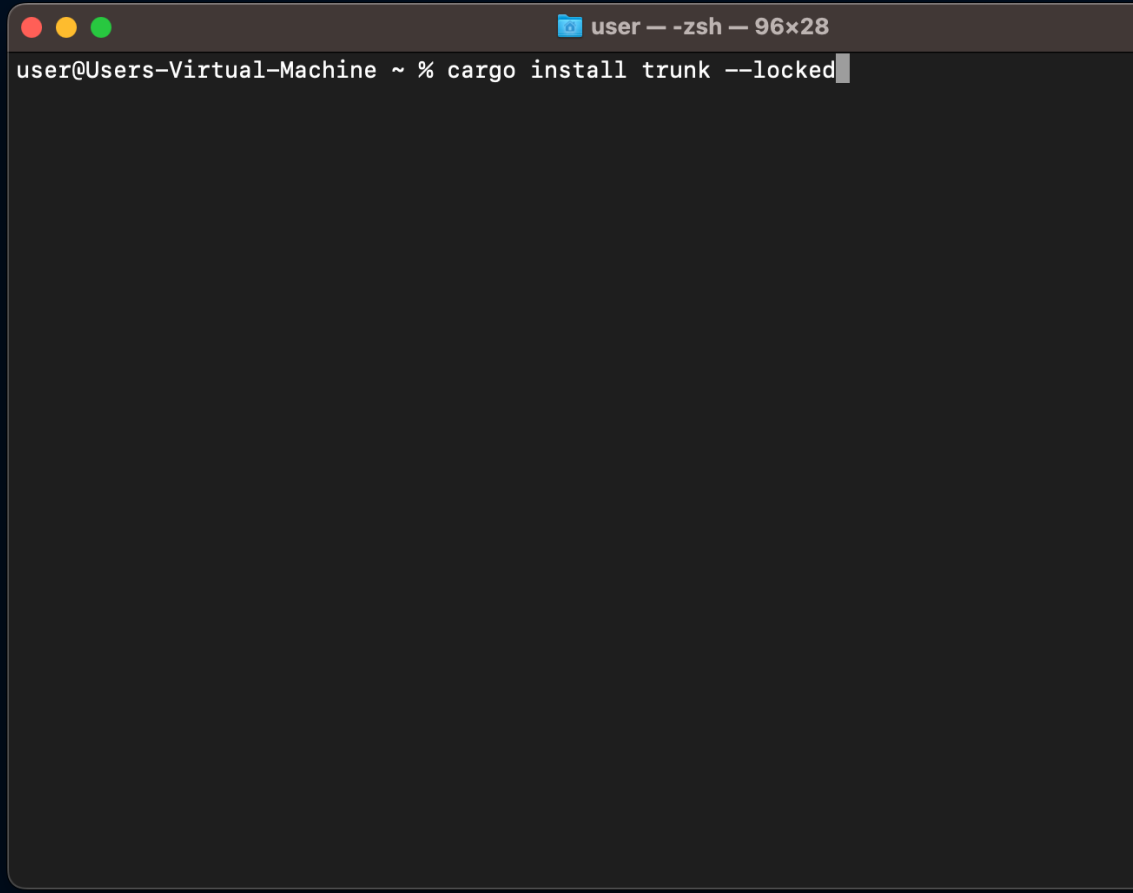
- Development Runner
- Build-tool and Packager
- Generate Icons
- Generate Signing Keys



CrabNebula

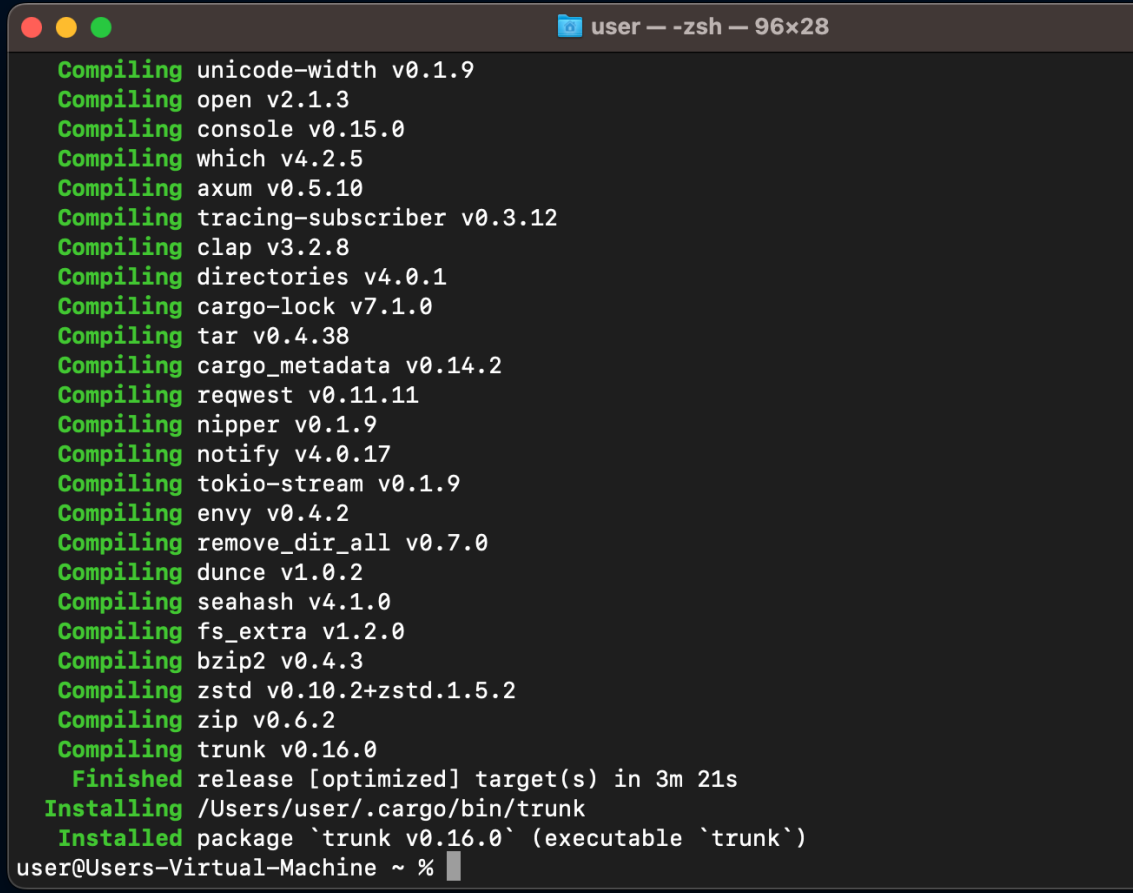
Trunk

- WASM web app bundler



Trunk

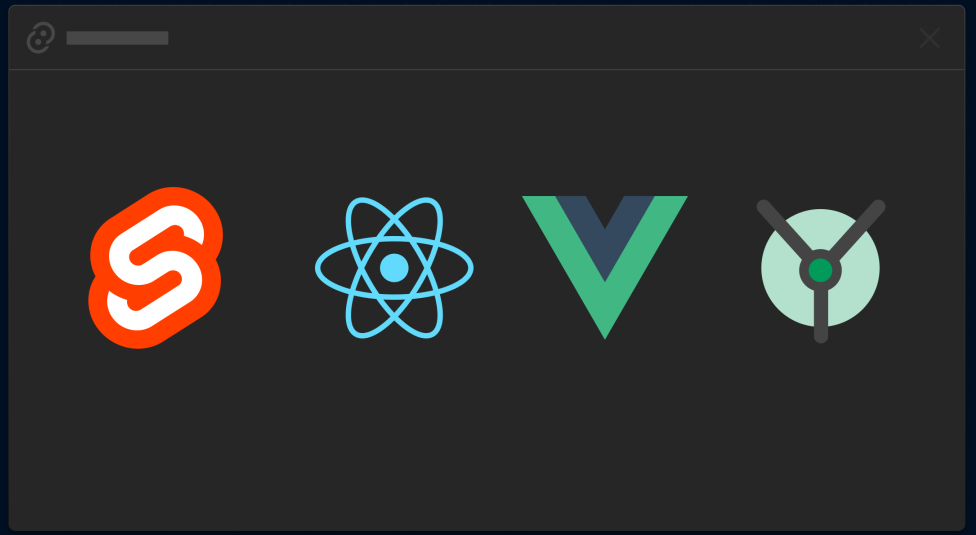
- WASM web app bundler



That Tauri thing; How does it work?

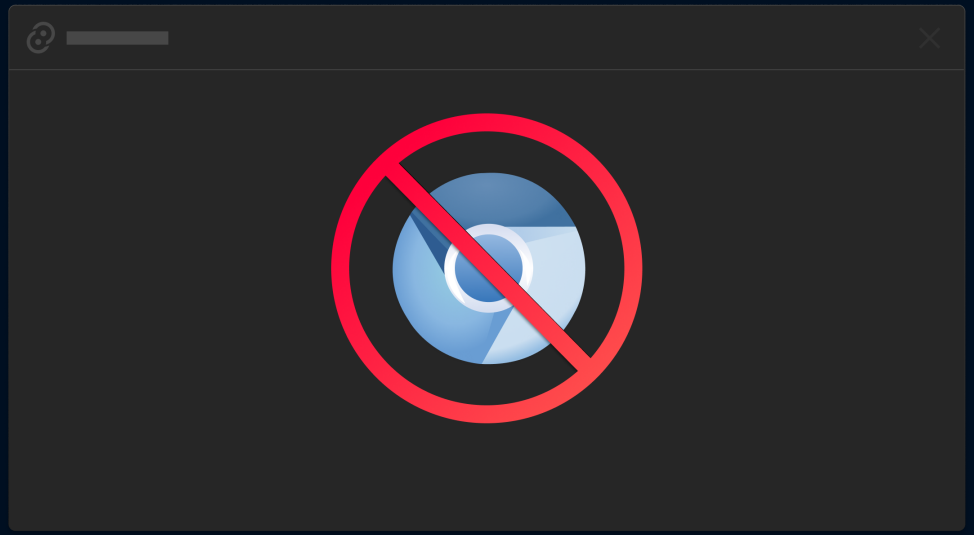
How does Tauri work?

- Renders HTML/CSS/JS Frontend



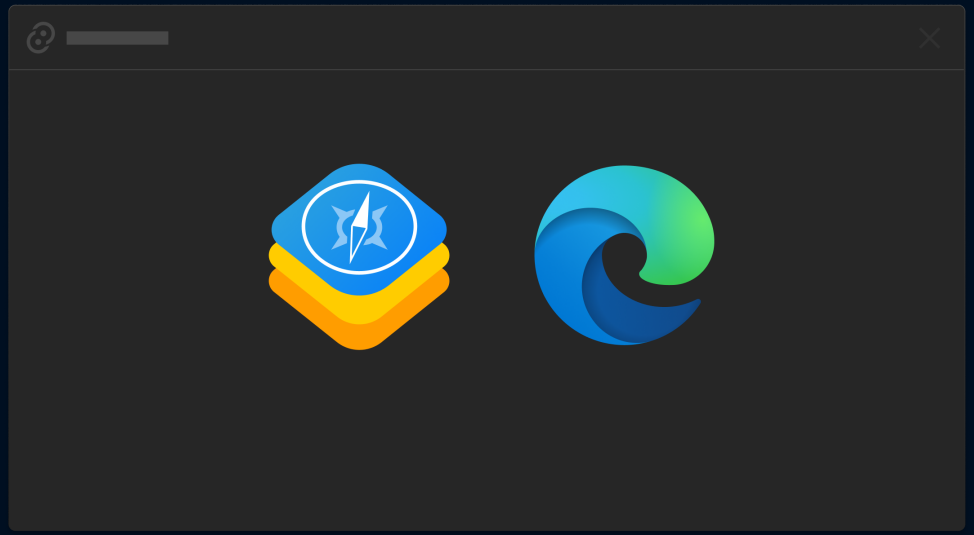
How does Tauri work?

- Renders HTML/CSS/JS Frontend
- No embedded browser



How does Tauri work?

- Renders HTML/CSS/JS Frontend
- No embedded browser
- Uses the OS WebView instead



How does Tauri work?

- WebView is Sandboxed
- No Access to Operating System
- CORS applies

How does Tauri work?

How does Tauri work?

How does Tauri work?

Inter-Process-Communication

TAO

Window Creation Library

- Windows
- Menus
- System Trays

CrabNebula

WRY

Webview Rendering Library

- Bindings to the OS Webview

- Windows: Edge WebView2
- macOS: WKWebView
- Linux: WebKitGTK

CrabNebula

TAURI

- Integrates Frontend Framework
- Additional APIs

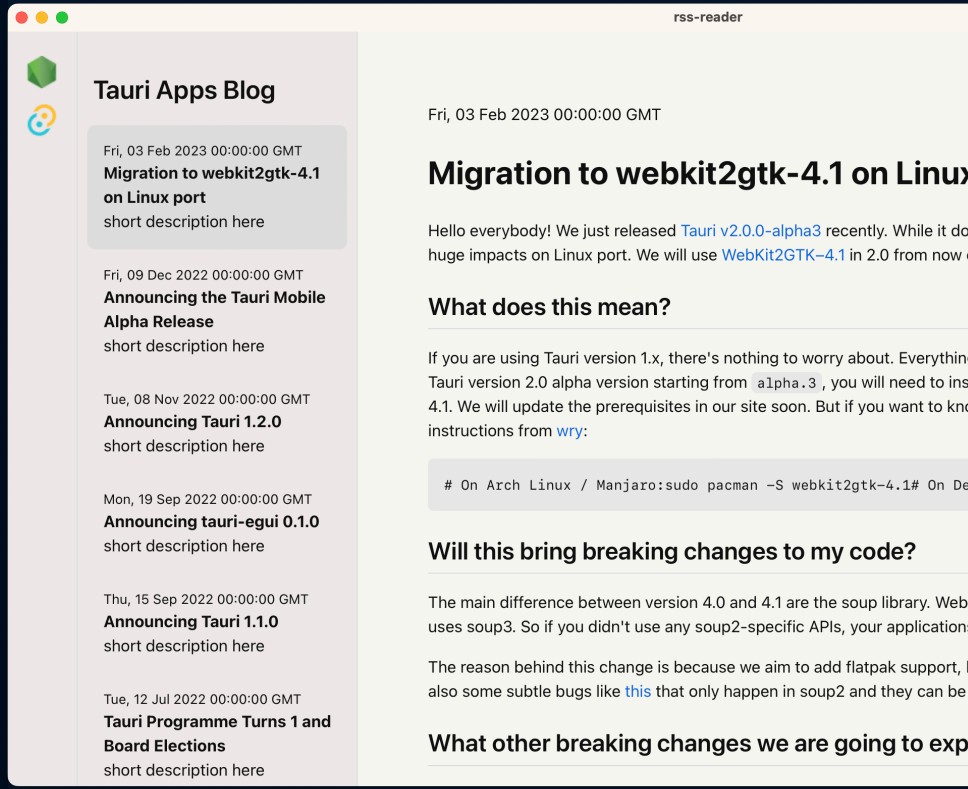
CrabNebula

TAURI

- Integrates Frontend Framework
- Additional APIs
- Bundling & Packaging

What are we building?

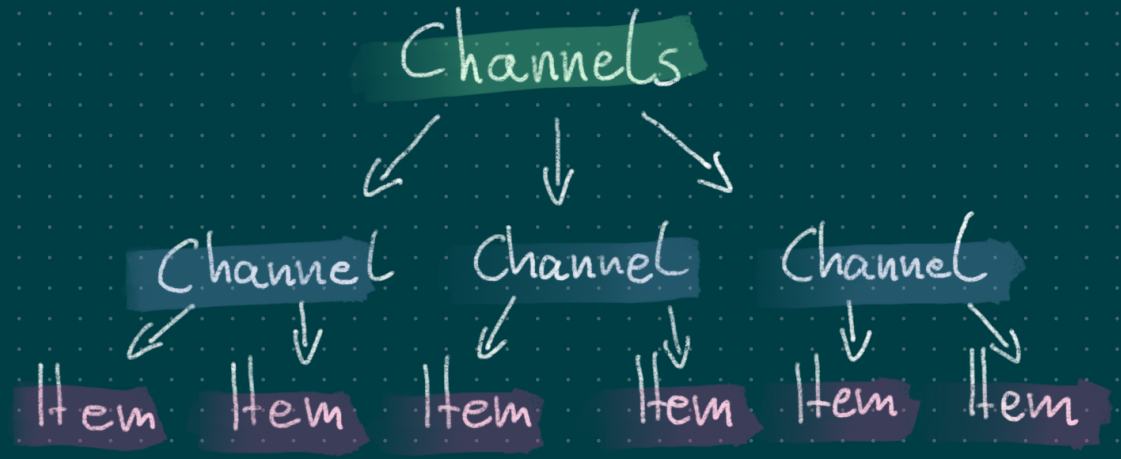
Yet another RSS reader!

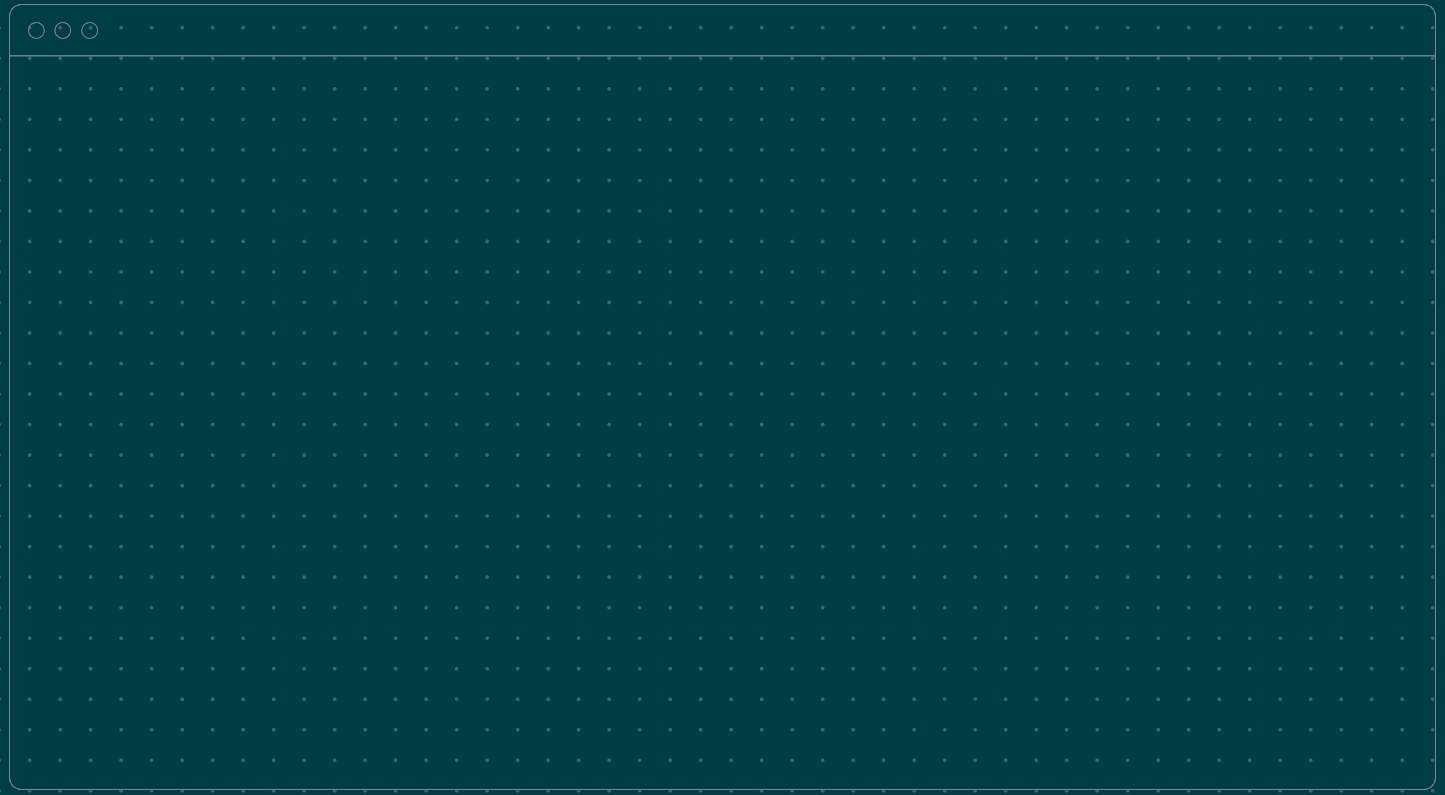


CrabNebula

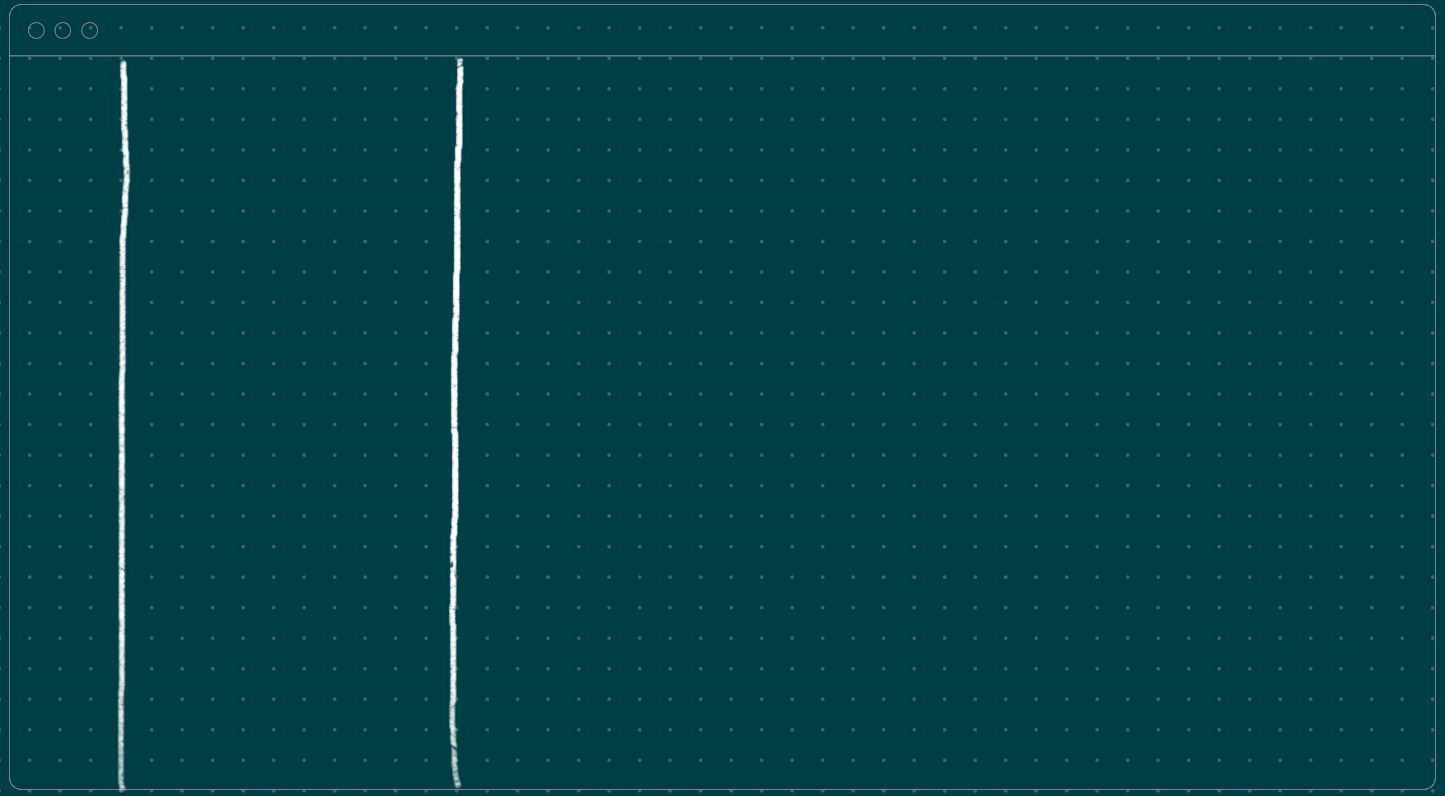
Channel


Channel
Item
Item
Item


Channels
Channel
Channel
Channel
Item
Item
Item
Item
Item
Item

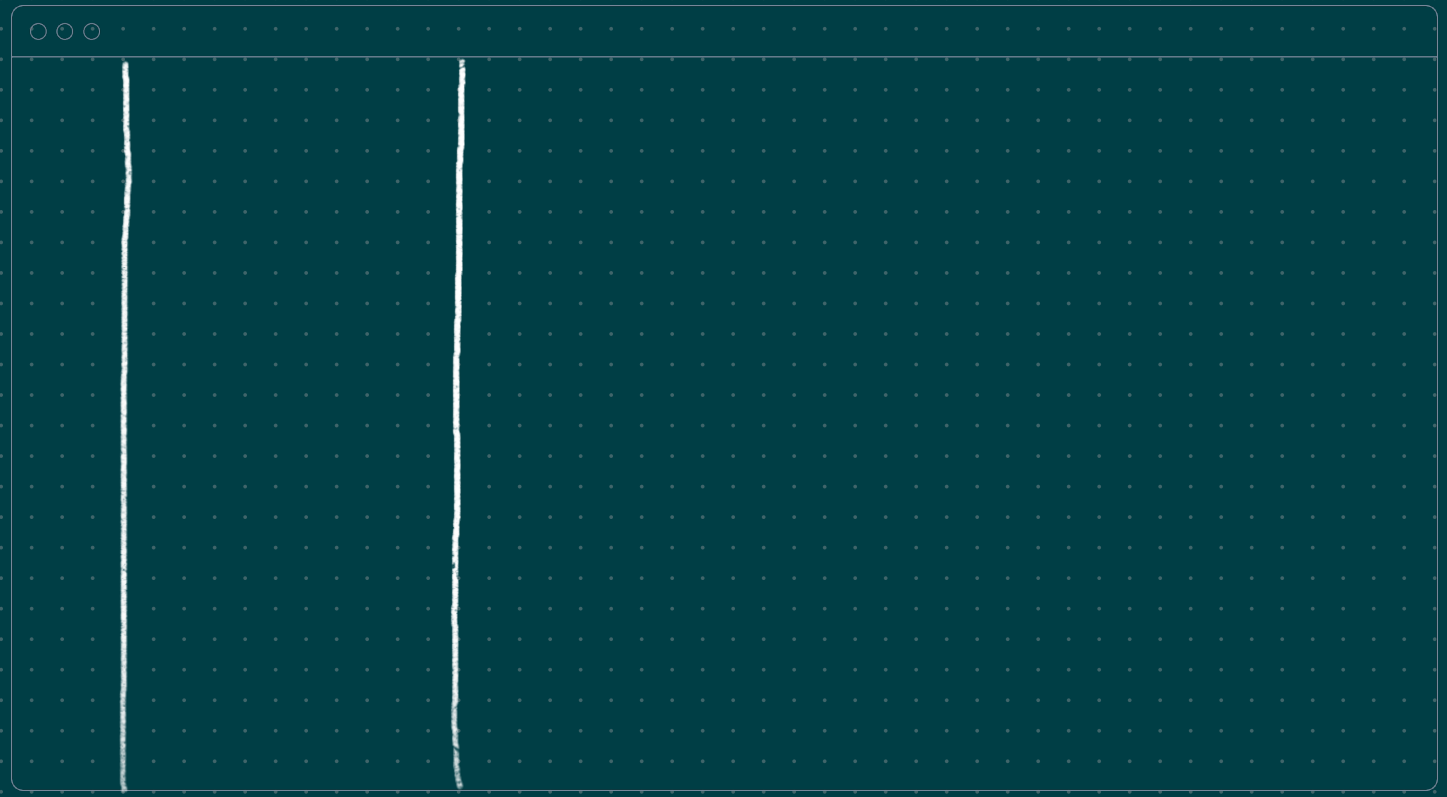


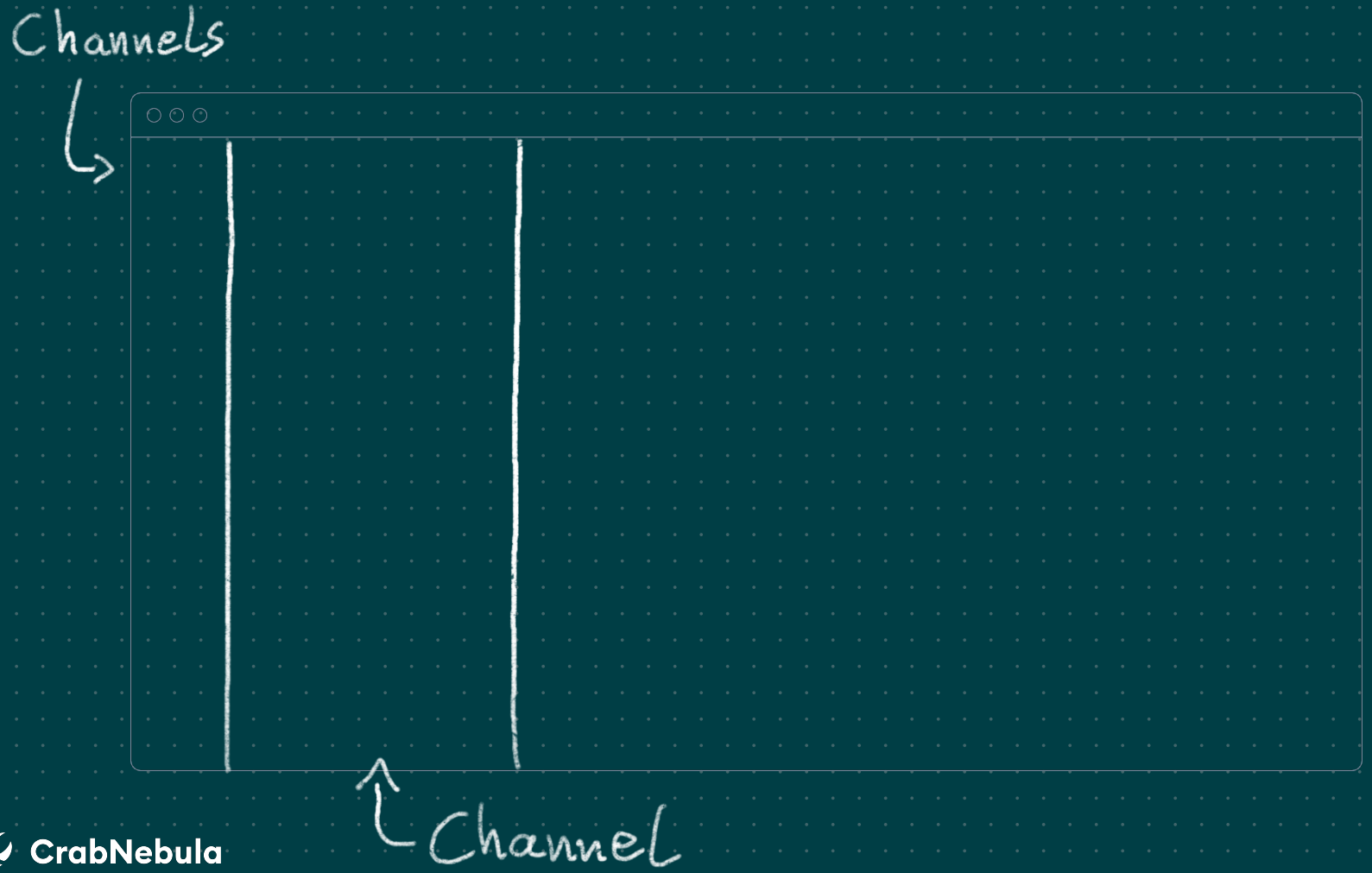
CrabNebula

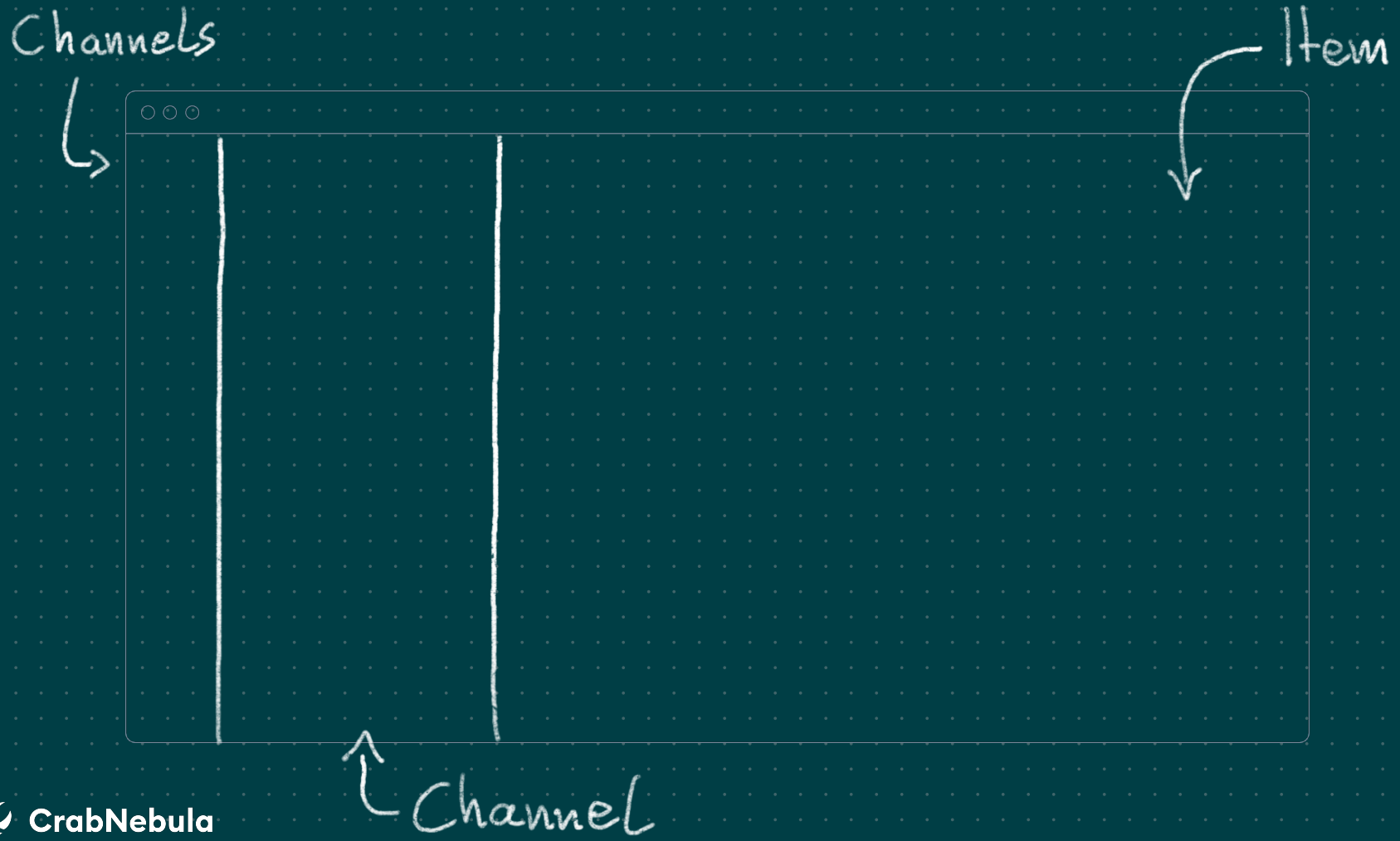


CrabNebula

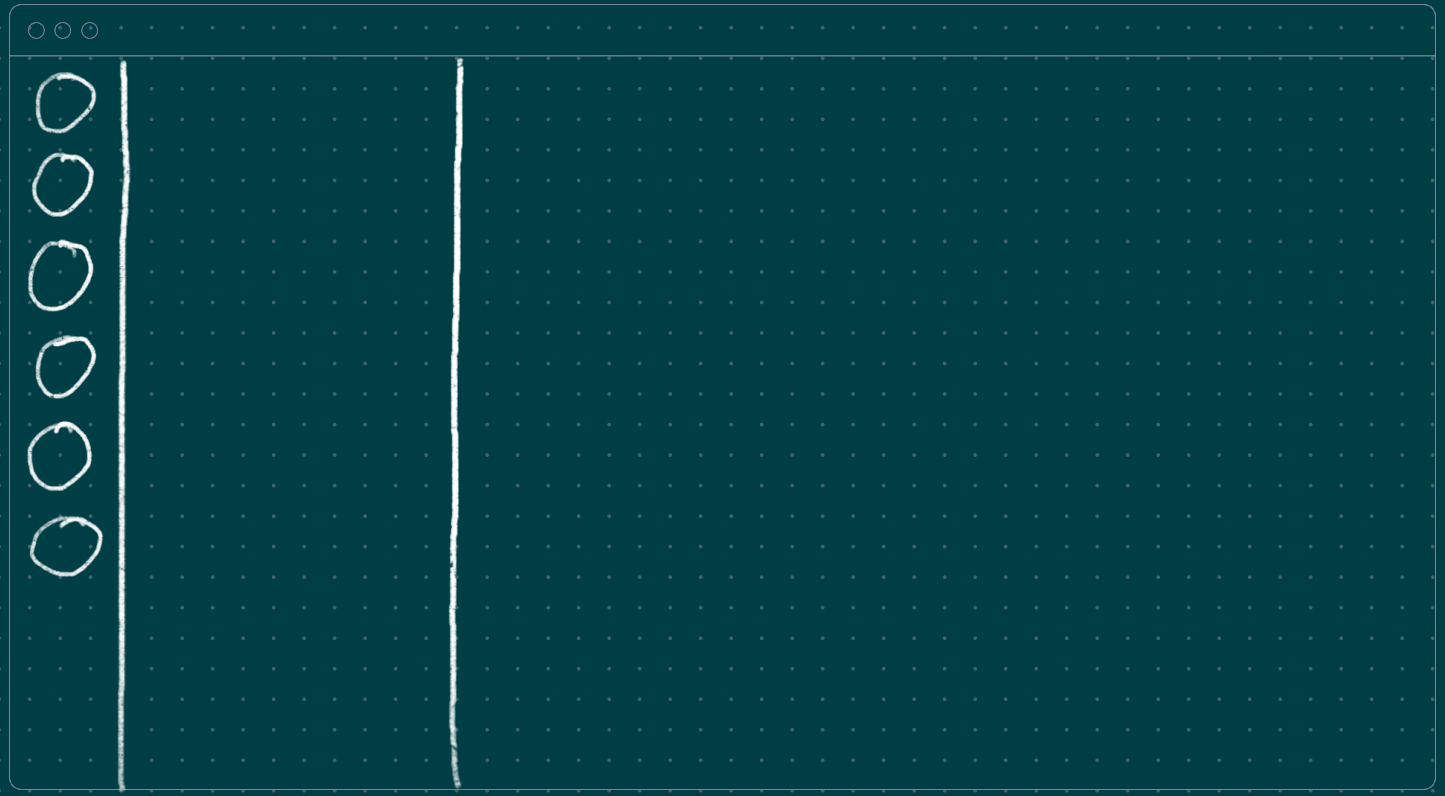
Channels



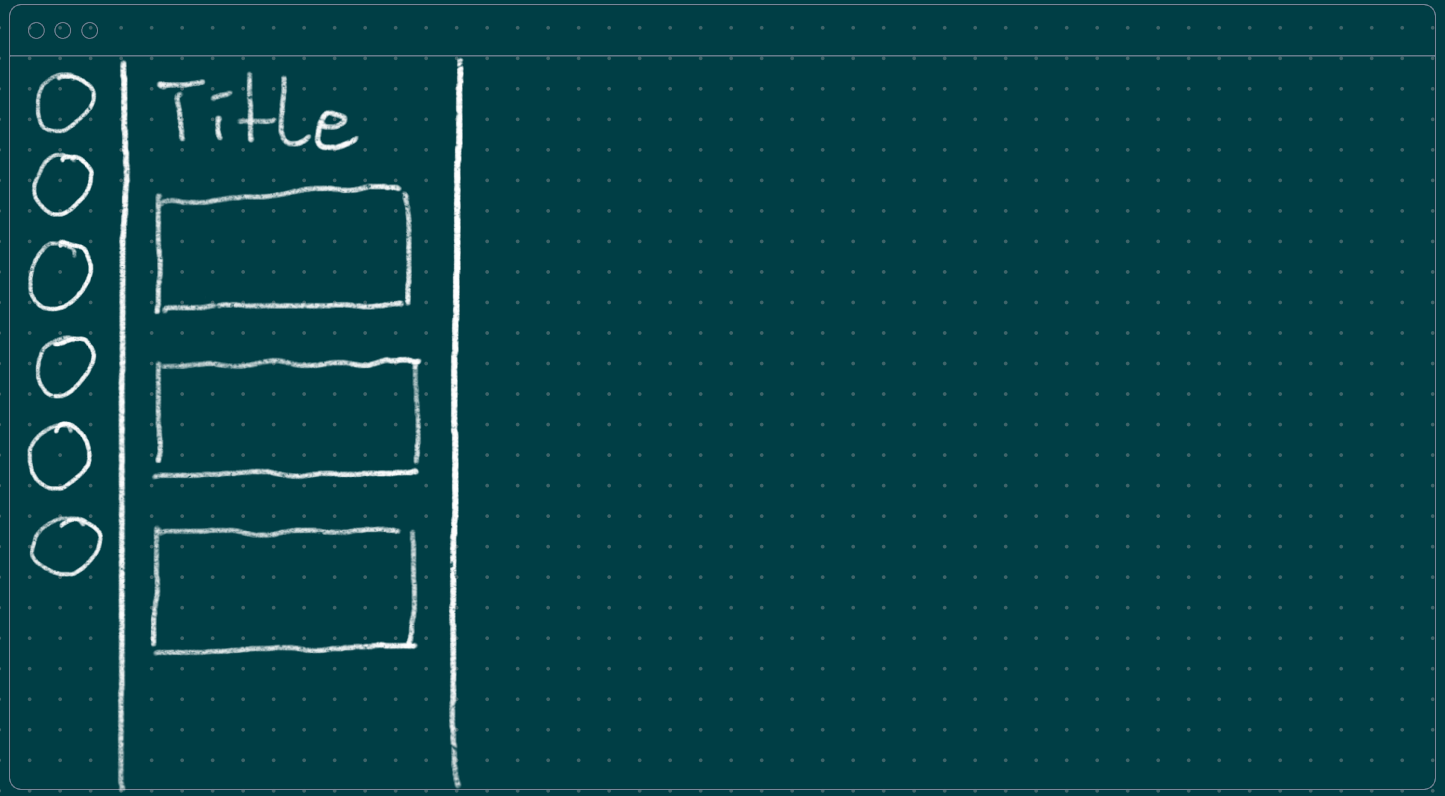

Channels
Channel

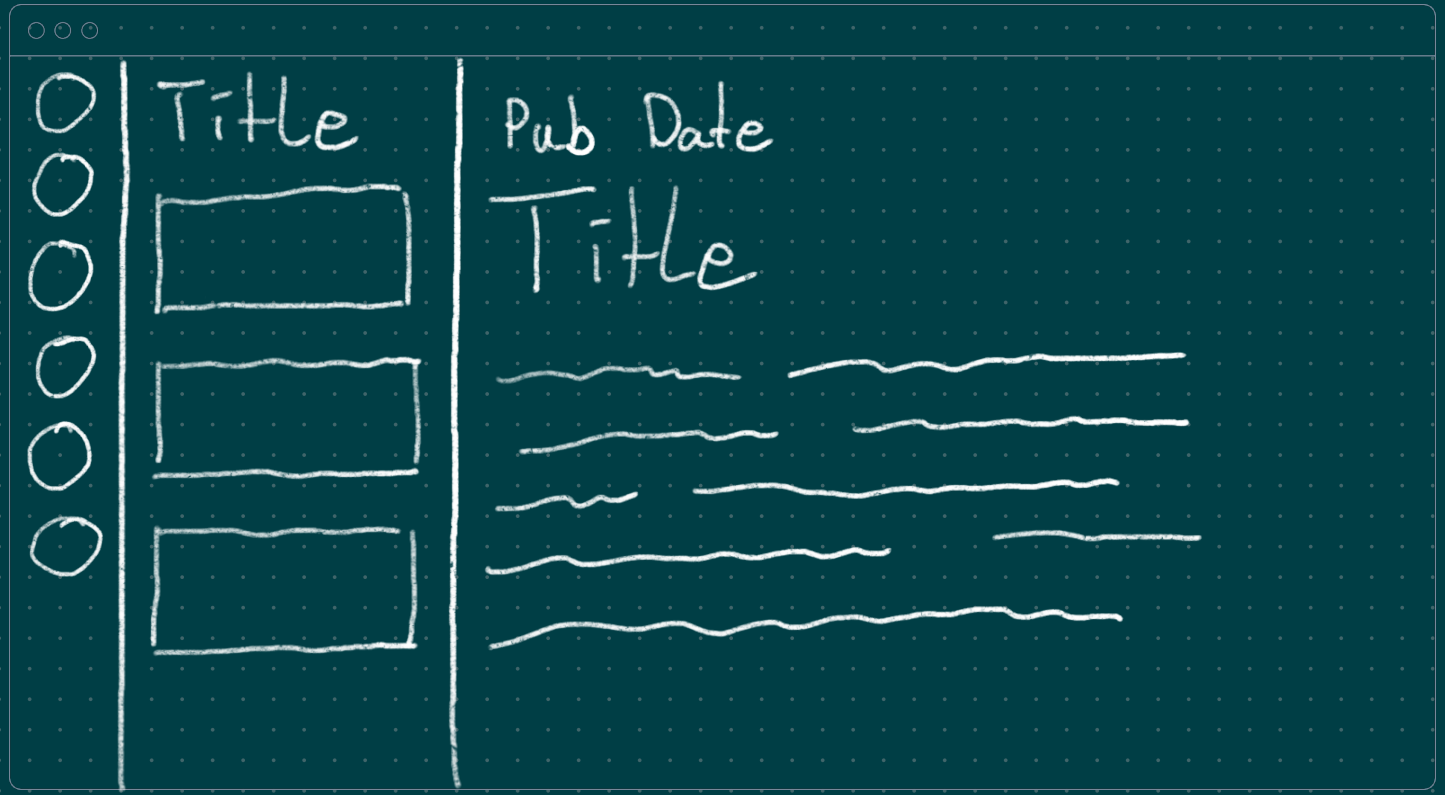


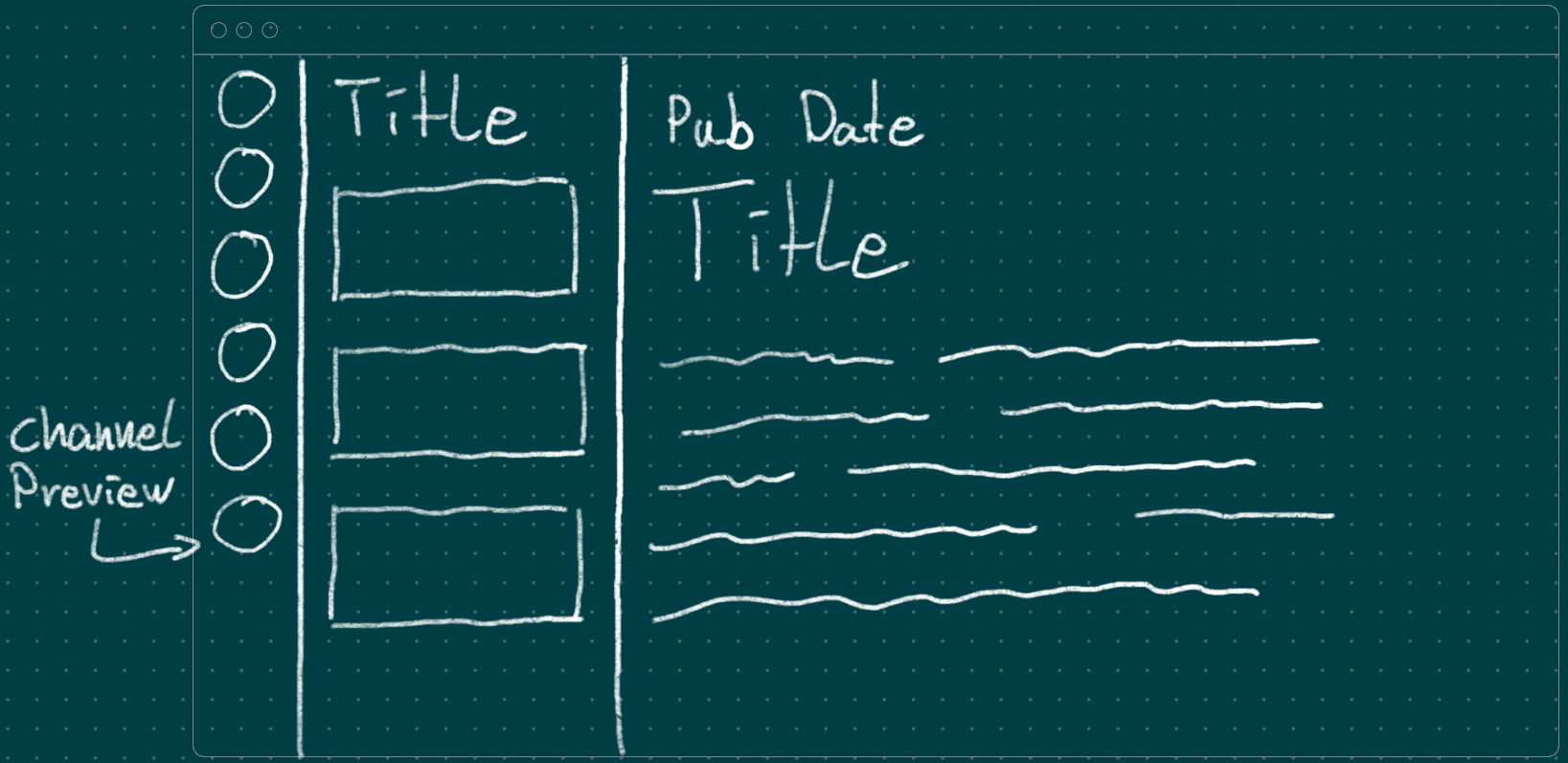
CrabNebula

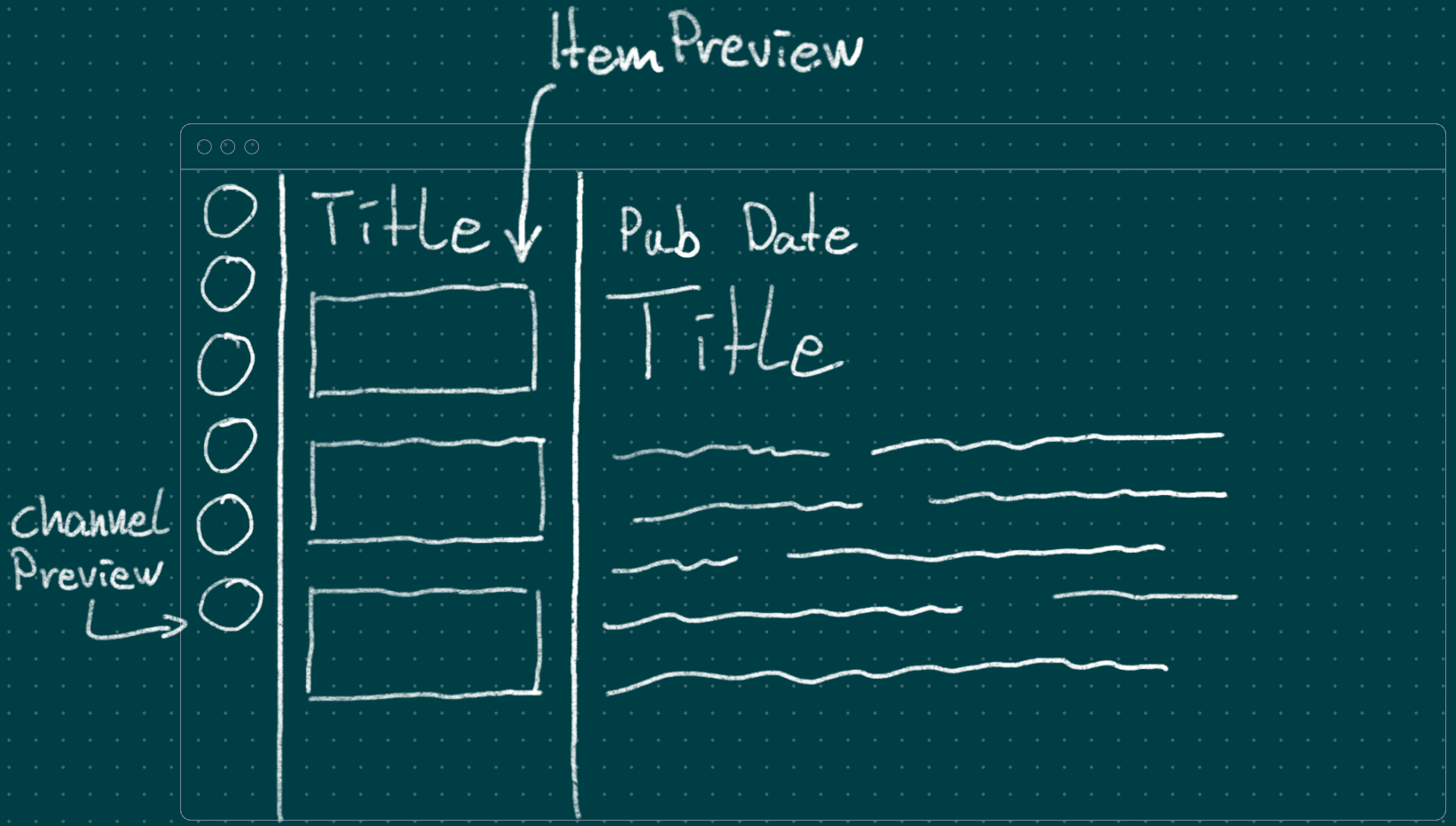


CrabNebula

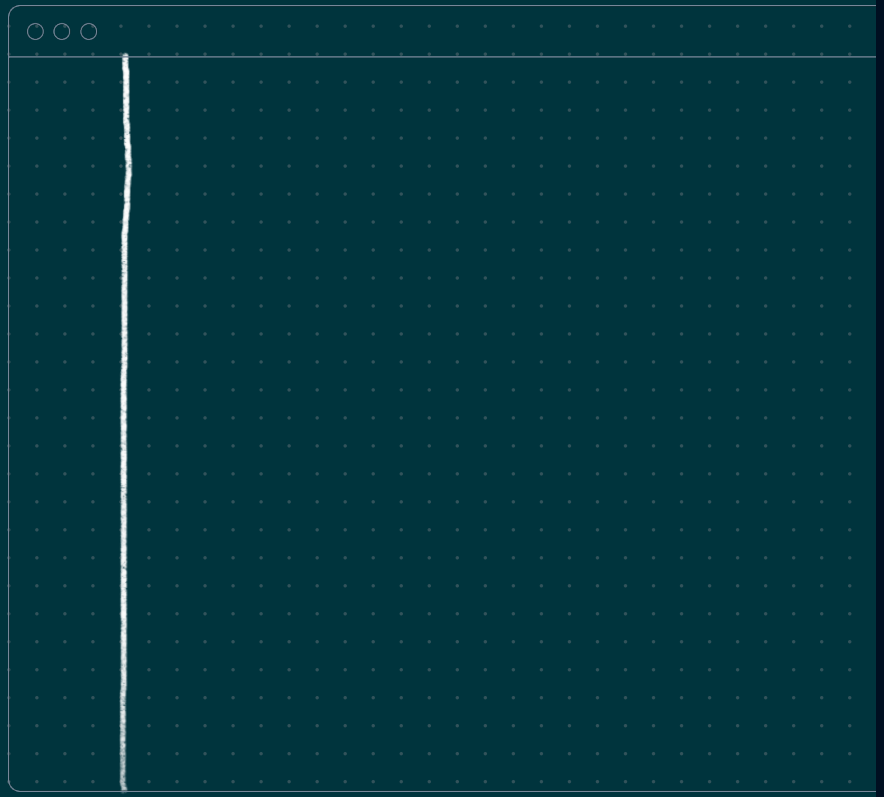

Title


Title
Pub Date
Title


Title
Pub Date
Title
Channel Preview


Item Preview
Title
Pub Date
Title
Channel
Preview

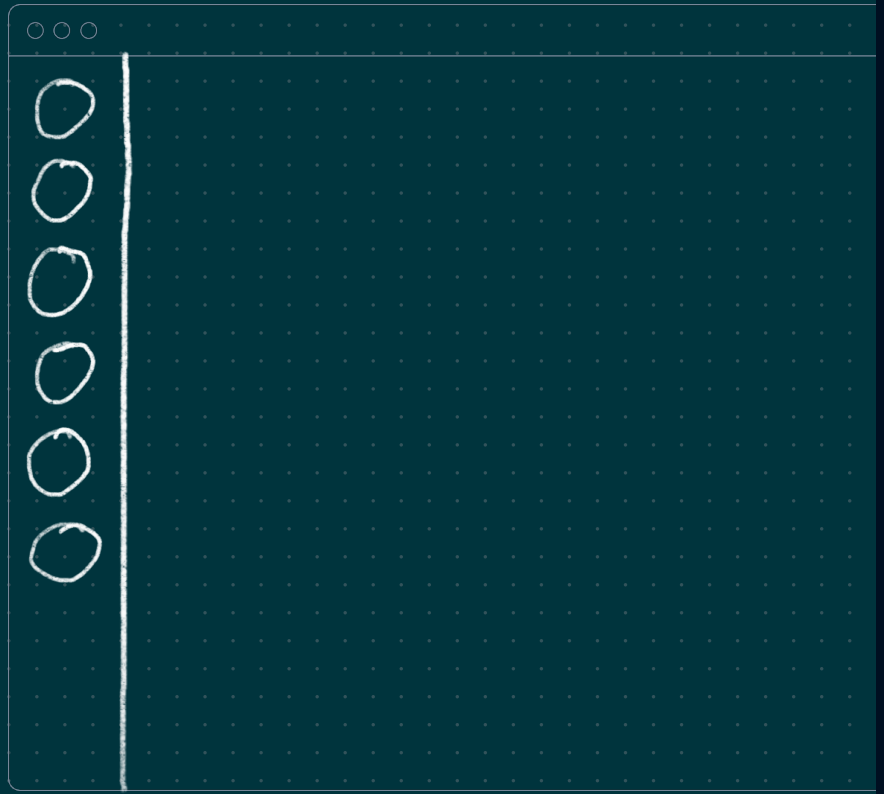
Channel List



src/components/channel_list.rs

```rust
use yew::prelude::*;

#[function_component]
pub fn ChannelList() -> Html {
    html! {
        <aside>
            <ul class="channels">
                {"channels"}
            </ul>
        </aside>
    }
}
```

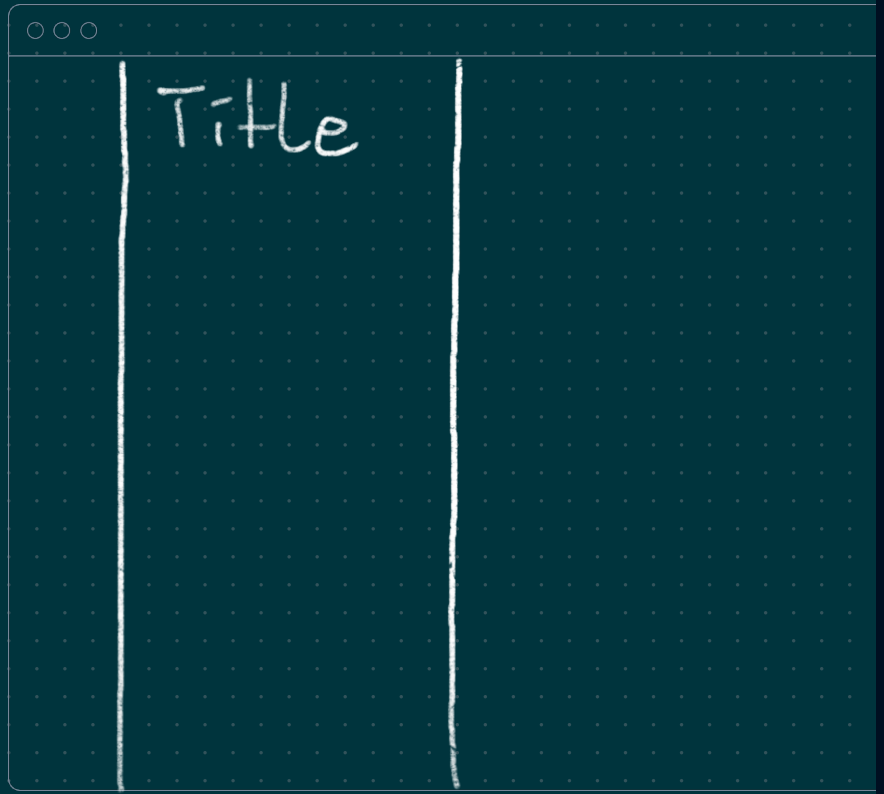
ChannelPreview



src/components/channel_preview.rs

```rust
#[function_component]
pub fn ChannelPreview() -> Html {
    html! {
        <li class="channel-preview">
            {"channel icon here"}
        </li>
    }
}
```

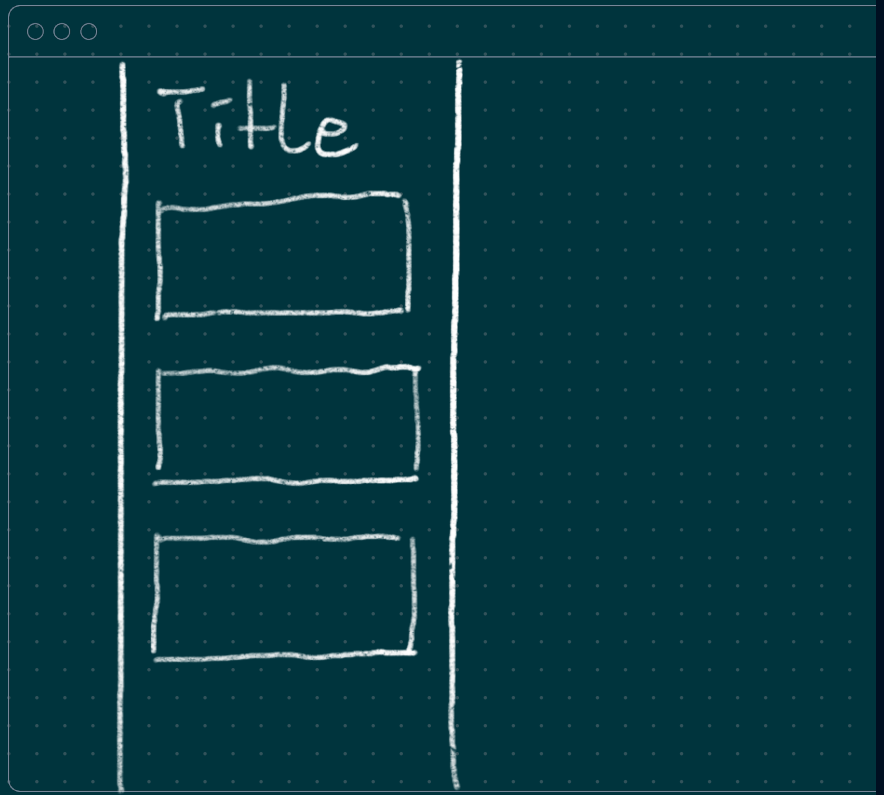
Channel



src/components/channel.rs

```
#[function_component]
pub fn Channel() -> Html {
    html! {
        <aside class="channel">
            <strong class="channel-title">
                {"title"}
            </strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    }
}
```

Channel



src/components/channel.rs

```
#[function_component]
pub fn Channel() -> Html {
    html! {
        <aside class="channel">
            <strong class="channel-title">
                {"title"}
            </strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    }
}
```

ItemPreview

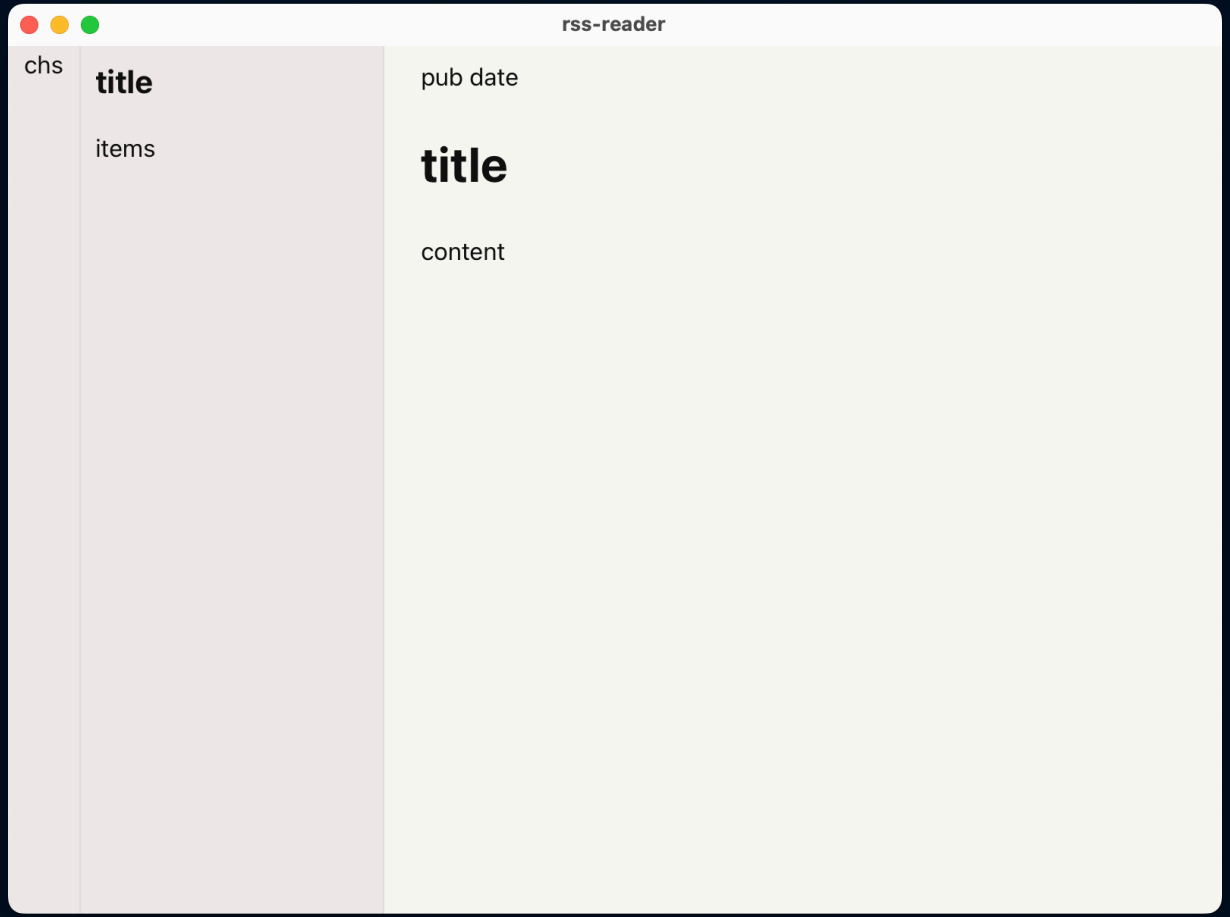
src/components/item_preview.rs

```
#[function_component]
pub fn ItemPreview() -> Html {
    html! {
        <li class="item-preview">
            {"pub date"}
            <strong>
                {"title"}
            </strong>
            <span class="item-preview-
description">
                {"short description"}
            </span>
        </li>
    }
}
```

Item

src/components/item.rs

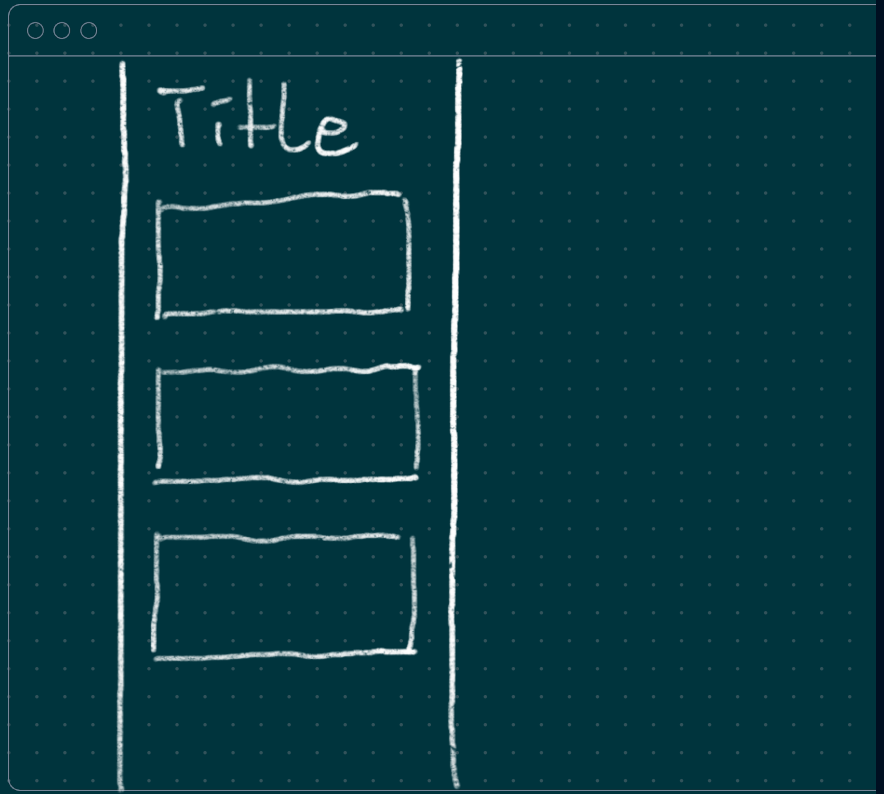
```
#[function_component]
pub fn Item() -> Html {
    html! {
        <main>
            <header>
                {"pub date"}
                <h1>{"title"}</h1>
            </header>
            <article>
                {"content"}
            </article>
        </main>
    }
}
```



CrabNebula

Application Logic

Channel



- Need to fetch the URL
- Parse the RSS Channel
- Render the title
- Render ItemPreviews

src-tauri/src/main.rs

```rust
async fn fetch_channel() {

}
```

src-tauri/src/main.rs

```rust
#[tauri::command]
async fn fetch_channel() {

}
```

src-tauri/src/main.rs

```rust
use url::Url;

#[tauri::command]
async fn fetch_channel(url: Url) {

}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {

}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);
}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{ClientBuilder, HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;
    let content = client.send(request).await?.bytes().await?;
}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{ClientBuilder, HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;
    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;
}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{ClientBuilder, HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;
    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;

    Ok(channel)
}
```

src-tauri/src/main.rs

```rust
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;
    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;

    Ok(channel)
}

fn main() {
    tauri::Builder::default()
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

src-tauri/src/main.rs

```rust
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;
    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;

    Ok(channel)
}

fn main() {
    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![fetch_channel])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

```
error[E0599]: the method `async_kind` exists for reference `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>`, but its
trait bounds were not satisfied
  --> src-tauri/src/main.rs:42:1
   |
42 | #[tauri::command]
   | ^^^^^^^^^^^^^^^^^ method cannot be called on `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>` due to
unsatisfied trait bounds
   ...
60 |         .invoke_handler(tauri::generate_handler![fetch_channel])
   |                         ------------------------------------- in this macro invocation
   |
   = note: the following trait bounds were not satisfied:
           `impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: serde::ser::Serialize`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::SerializeKind`
           `<&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>> as Future>::Output = Result<_, _>`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::ResultFutureKind`
           `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: Future`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::ResultFutureKind`
   = note: this error originates in the macro `__cmd__fetch_channel` which comes from the expansion of the macro `tauri::generate_handler` (in Nightly builds,
run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0599`.
error: could not compile `rss-reader` due to previous error
```

CrabNebula

```
error[E0599]: the method `async_kind` exists for reference `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>`, but its
trait bounds were not satisfied
  --> src-tauri/src/main.rs:42:1
   |
42 | #[tauri::command]
   | ^^^^^^^^^^^^^^^^^ method cannot be called on `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>` due to
unsatisfied trait bounds
   ...
60 |         .invoke_handler(tauri::generate_handler![fetch_channel])
   |                         ------------------------------------- in this macro invocation
   |
   = note: the following trait bounds were not satisfied:
           `impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: serde::ser::Serialize`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::SerializeKind`
           `<&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>> as Future>::Output = Result<_, _>`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::ResultFutureKind`
           `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: Future`
           which is required by `&impl Future<Output = Result<Channel, Box<(dyn std::error::Error + 'static)>>>: tauri::command::private::ResultFutureKind`
   = note: this error originates in the macro `__cmd__fetch_channel` which comes from the expansion of the macro `tauri::generate_handler` (in Nightly builds,
run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0599`.
error: could not compile `rss-reader` due to previous error
```

CrabNebula

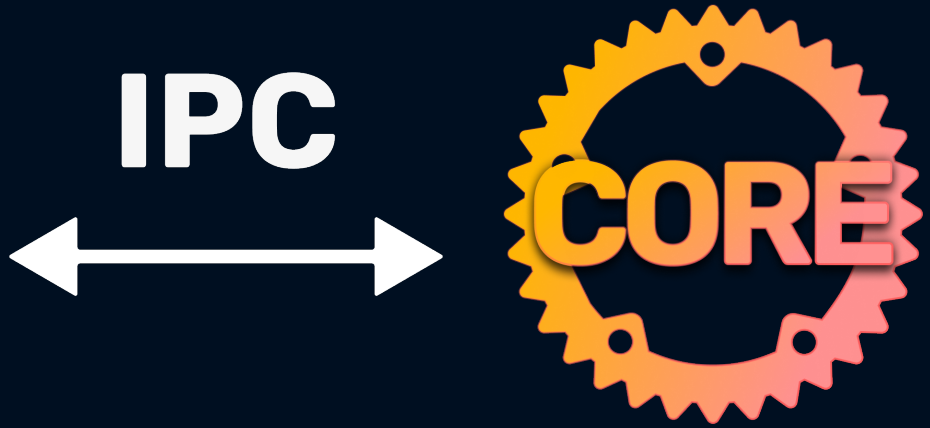
src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{ClientBuilder, HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;

    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;

    Ok(channel)
}
```

src-tauri/src/main.rs

```rust
use url::Url;
use rss::Channel;
use tauri::api::http::{ClientBuilder, HttpRequestBuilder, ResponseType};

#[tauri::command]
async fn fetch_channel(url: Url) -> Result<Channel, Box<dyn std::error::Error>> {
    let request = HttpRequestBuilder::new("GET", url)?
        .response_type(ResponseType::Binary);

    let client = ClientBuilder::new().build()?;

    let content = client.send(request).await?.bytes().await?;

    let channel = Channel::read_from(&content.data[..])?;

    Ok(channel)
}
```

Commands



All types MUST implement serde::Serialize !

src-tauri/src/main.rs

```rust
#[derive(Debug, thiserror::Error)]
enum Error {
    #[error(transparent)]
    Tauri(#[from] tauri::api::Error),
    #[error(transparent)]
    Rss(#[from] rss::Error)
}
```

src-tauri/src/main.rs

```rust
#[derive(Debug, thiserror::Error)]
enum Error {
    #[error(transparent)]
    Tauri(#[from] tauri::api::Error),
    #[error(transparent)]
    Rss(#[from] rss::Error)
}

impl serde::Serialize for Error {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::ser::Serializer,
    {
        serializer.serialize_str(self.to_string().as_ref())
    }
}
```

src/component/channel.rs

```rust
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(catch, js_namespace = ["window", "__TAURI__", "tauri"])]
    async fn invoke(cmd: &str, args: JsValue) -> Result<JsValue, JsValue>;
}
```

src/component/channel.rs

```rust
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(catch, js_namespace = ["window", "__TAURI__", "tauri"])]
    async fn invoke(cmd: &str, args: JsValue) -> Result<JsValue, JsValue>;
}

#[derive(Serialize)]
struct FetchChannelArgs<'a> {
    url: &'a Url,
}
```

src/component/channel.rs

```rust
#[function_component]
pub fn Channel(props: &ChannelProps) -> HtmlResult {
```

src/component/channel.rs

```rust
#[function_component]
pub fn Channel(props: &ChannelProps) -> HtmlResult {
    let result = use_future_with_deps(
        |url| async move {
        },
        props.url.clone(),
    )?;
```

src/component/channel.rs

```rust
#[function_component]
pub fn Channel(props: &ChannelProps) -> HtmlResult {
    let result = use_future_with_deps(
        |url| async move {
            let raw_args = serde_wasm_bindgen::to_value(
                &FetchChannelArgs { url: &url }
            )?;
        },
        props.url.clone(),
    )?;
```

src/component/channel.rs

```rust
#[function_component]
pub fn Channel(props: &ChannelProps) -> HtmlResult {
    let result = use_future_with_deps(
        |url| async move {
            let raw_args = serde_wasm_bindgen::to_value(
                &FetchChannelArgs { url: &url }
            )?;
            let raw_result = invoke("fetch_channel", raw_args).await?;
        },
        props.url.clone(),
    )?;
```

src/component/channel.rs

```rust
#[function_component]
pub fn Channel(props: &ChannelProps) -> HtmlResult {
    let result = use_future_with_deps(
        |url| async move {
            let raw_args = serde_wasm_bindgen::to_value(
                &FetchChannelArgs { url: &url }
            )?;
            let raw_result = invoke("fetch_channel", raw_args).await?;
            serde_wasm_bindgen::from_value::<rss::Channel>(raw_result)
        },
        props.url.clone(),
    )?;
```

src/component/channel.rs

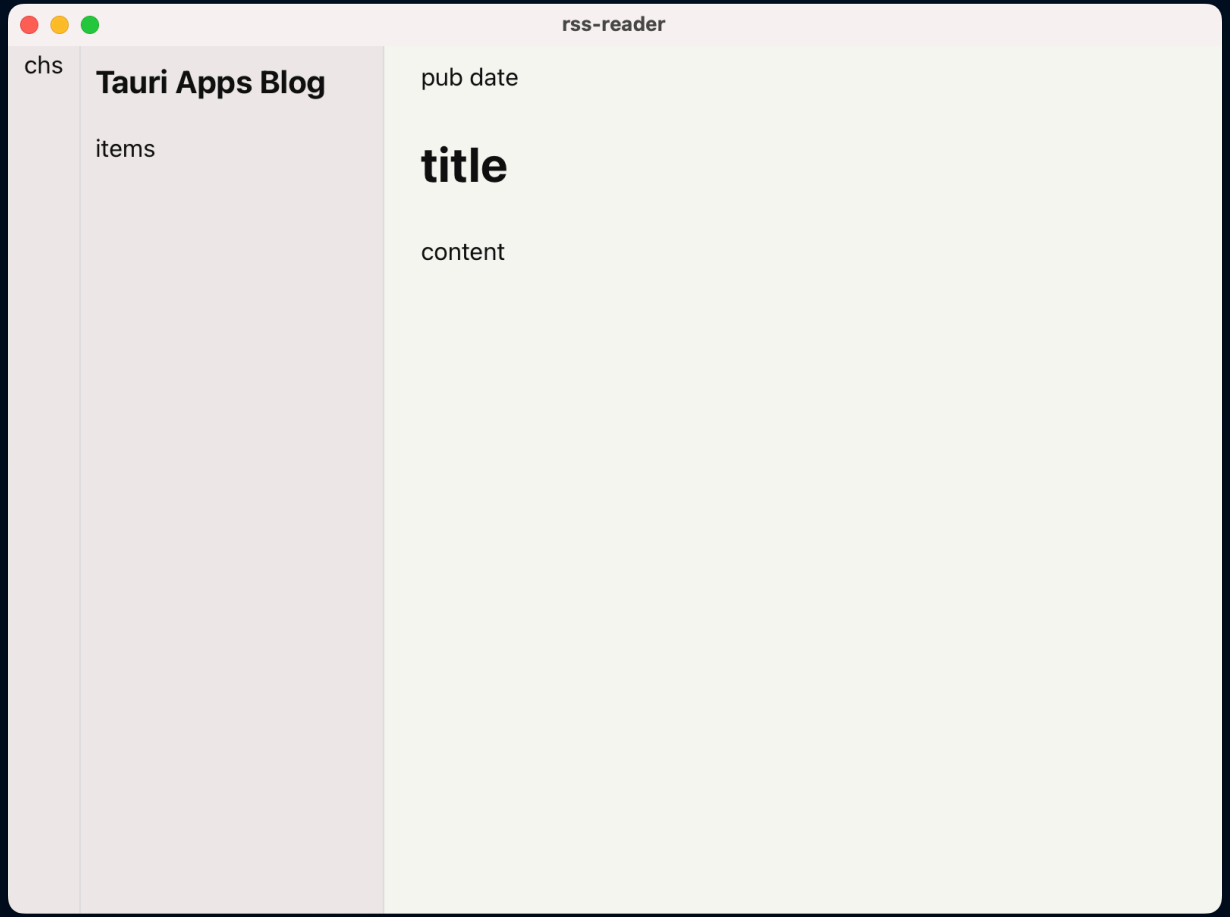
```rust
pub fn Channel(props: &ChannelProps) -> HtmlResult {
    let result = use_future_with_deps(
        |url| async move {
            let raw_args = serde_wasm_bindgen::to_value(
                &FetchChannelArgs { url: &url }
            )?;
            let raw_result = invoke("fetch_channel", raw_args).await?;
            serde_wasm_bindgen::from_value::<rss::Channel>(raw_result)
        },
        props.url.clone(),
    )?;

    let Ok(channel) = &*result else {
        return Ok(html! { "Failed to fetch channel" });
    };
```

src/component/channel.rs

```
        props.url.clone(),
    )?;

    let Ok(channel) = &*result else {
        return Ok(html! { "Failed to fetch channel" });
    };

    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{"title"}</strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    })
```

src/component/channel.rs

```
        props.url.clone(),
    )?;

    let Ok(channel) = &*result else {
        return Ok(html! { "Failed to fetch channel" });
    };

    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{&channel.title}</strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    })
```

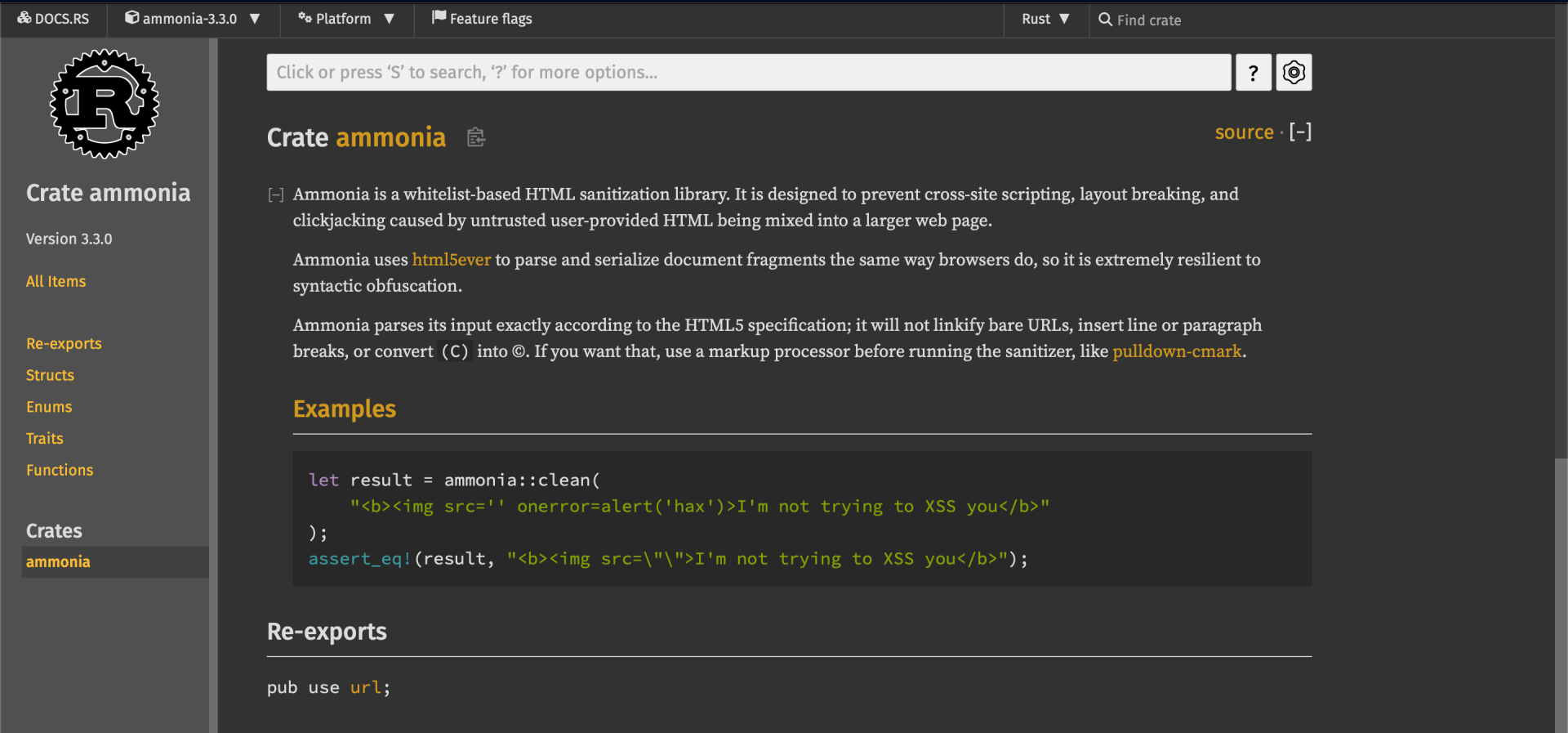


CrabNebula

src/component/item_preview.rs

```rust
#[function_component]
pub fn ItemPreview(props: &ItemPreviewProps) -> Html {
    let safe_description = props
        .description
        .as_ref()
        .map(|input| AmmoniaBuilder::empty().clean(input).to_string())
        .unwrap_or_default();

    html! {
        <li key={props.id} class="item-preview">
            {"pub date"}
            if let Some(title) = &props.title {
                <strong>
                    {title}
                </strong>
```

Description



src/component/item_preview.rs

```rust
#[function_component]
pub fn ItemPreview(props: &ItemPreviewProps) -> Html {
    let safe_description = props
        .description
        .as_ref()
        .map(|input| AmmoniaBuilder::empty().clean(input).to_string())
        .unwrap_or_default();

    html! {
        <li key={props.id} class="item-preview">
            {"pub date"}
            if let Some(title) = &props.title {
                <strong>
                    {title}
                </strong>
```

src/component/item_preview.rs

```
        .unwrap_or_default();

    html! {
        <li key={props.id} class="item-preview">
            {"pub date"}
            if let Some(title) = &props.title {
                <strong>
                    {title}
                </strong>
            }
            <span class="item-preview-description">
                {"short description here"}
            </span>
        </li>
    }
}
```

src/component/item_preview.rs

```
        .unwrap_or_default();

    html! {
        <li key={props.id} class="item-preview">
            {"pub date"}
            if let Some(title) = &props.title {
                <strong>
                    {title}
                </strong>
            }
            <span class="item-preview-description">
                {&safe_description[0..100.min(safe_description.len())]}
            </span>
        </li>
    }
}
```

src/component/channel.rs

```
    let Ok(channel) = &*result else {
        return Ok(html! { "Failed to fetch channel" });
    };

    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{&channel.title}</strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    })
```

src/component/channel.rs

```
    let items = channel
        .items
        .iter()
        .enumerate()
        .map(|(id, item)| {
        });


    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{&channel.title}</strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    })
```

src/component/channel.rs

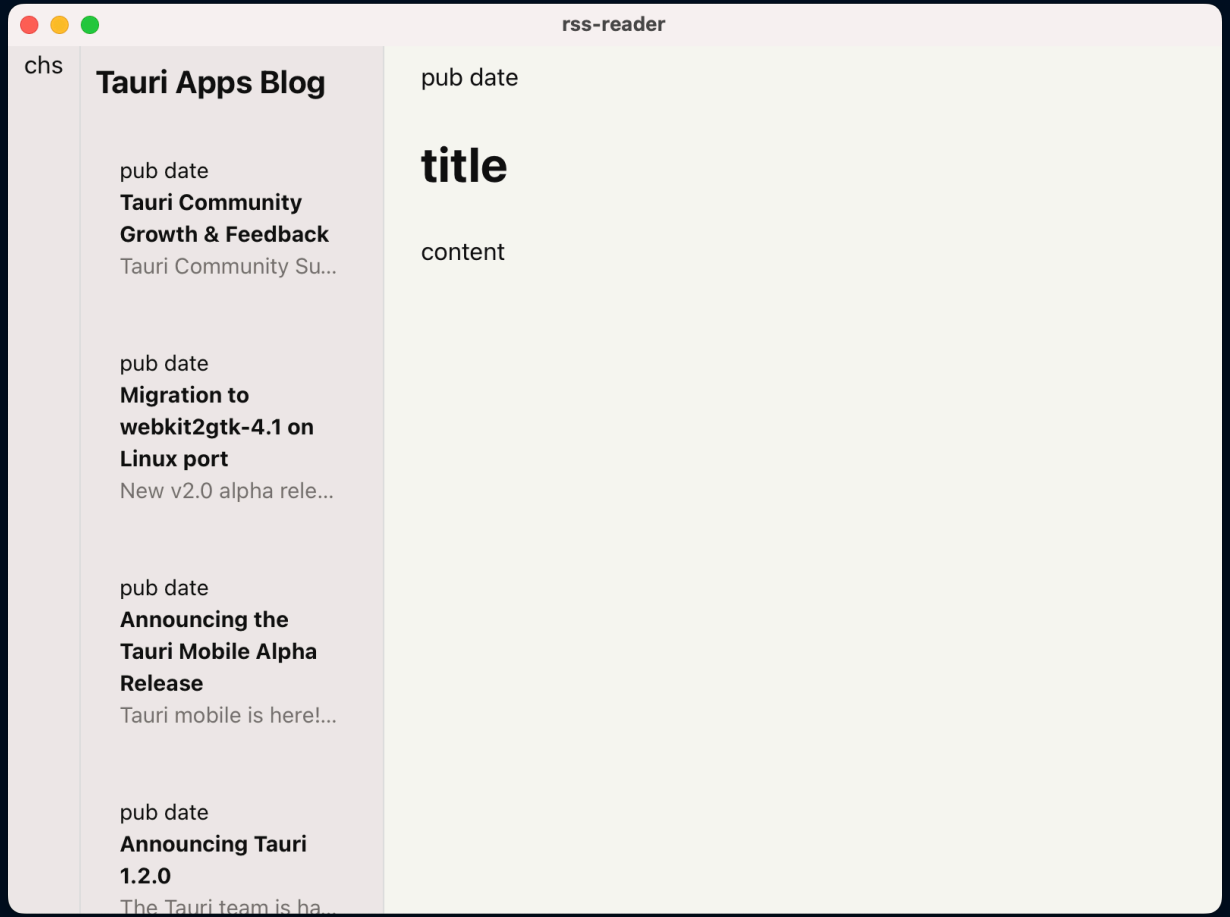
```
    let items = channel
        .items
        .iter()
        .enumerate()
        .map(|(id, item)| {
            html! {
                <ItemPreview
                    {id}
                    channel_url={props.url.clone()}
                    channel_title={channel.title.clone()}
                    title={item.title.clone()}
                    pub_date={item.pub_date.clone()}
                    description={item.description.clone()}
                />
            }
        });
```

src/component/channel.rs

```
                    channel_title {channel.title.clone()}
                    title={item.title.clone()}
                    pub_date={item.pub_date.clone()}
                    description={item.description.clone()}
                />
            }
        });


    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{&channel.title}</strong>
            <ul>
                {"items"}
            </ul>
        </aside>
    })
}
```

src/component/channel.rs

```
                    channel_title {channel.title.clone()}
                    title={item.title.clone()}
                    pub_date={item.pub_date.clone()}
                    description={item.description.clone()}
                />
            }
        });


    Ok(html! {
        <aside class="channel">
            <strong class="channel-title">{&channel.title}</strong>
            <ul>
                {items.collect::<Html>()}
            </ul>
        </aside>
    })
}
```


rss-reader
chs
Tauri Apps Blog
pub date
Tauri Community Growth & Feedback
Tauri Community Su...
pub date
Migration to webkit2gtk-4.1 on Linux port
New v2.0 alpha rele...
pub date
Announcing the Tauri Mobile Alpha Release
Tauri mobile is here!...
pub date
Announcing Tauri 1.2.0
The Tauri team is ha...
pub date
title
content

Takeaways

- Two-process model
- Sharing types between processes is great
- All types crossing the IPC bridge are serialized
- wasm-bindgen let’s us harness the power of JavaScript too
- Don’t be afraid of new Rust features!

Exercises for the reader

EVERY BYTE MATTERS