Your First Component

This chapter will teach you how to create a Component that displays a link to a post on hackernews.

Setup

> Before you start the guide, make sure you have the dioxus CLI and any required dependencies for your platform as described in the getting started guide.

First, let's create a new project for our hacker news app. We can use the CLI to create a new project. You can select a platform of your choice or view the getting started guide for more information on each option. If you aren't sure what platform to try out, we recommend getting started with web or desktop:

```
dx new
```
Copy

The template contains some boilerplate to help you get started. For this guide, we will be rebuilding some of the code from scratch for learning purposes. You can clear the `src/main.rs` file. We will be adding new code in the next sections.

Next, let's setup our dependencies. We need to set up a few dependencies to work with the hacker news API:

```
cargo add chrono --features serde
cargo add futures
cargo add reqwest --features json
cargo add serde --features derive
cargo add serde_json
cargo add async_recursion
```
Copy

Describing the UI

Now, we can define how to display a post. Dioxus is a *declarative* framework. This means that instead of telling Dioxus what to do (e.g. to "create an element" or "set the color to red") we simply *declare* how we want the UI to look.

To declare what you want your UI to look like, you will need to use the `rsx` macro. Let's create a `main` function and an `App` component to show information about our story:

```
fn main() {
    launch(App);
}

pub fn App() -> Element {
    rsx! {"story"}
}
```
Copy

Now if you run your application you should see something like this:

story

> RSX mirrors HTML. Because of this you will need to know some html to use Dioxus.
>
> Here are some resources to help get you started learning HTML:
>
> - MDN HTML Guide
> - W3 Schools HTML Tutorial
>
> In addition to HTML, Dioxus uses CSS to style applications. You can either use traditional CSS (what this guide uses) or use a tool like tailwind CSS:
>
> - MDN Traditional CSS Guide
> - W3 Schools Traditional CSS Tutorial
> - Tailwind tutorial (used with the Tailwind setup example)
>
> If you have existing html code, you can use the translate command to convert it to RSX. Or if you prefer to write html, you can use the html! macro to write html

> directly in your code.

Dynamic Text

Let's expand our `App` component to include the story title, author, score, time posted, and number of comments. We can insert dynamic text in the render macro by inserting variables inside `{}`s (this works similarly to the formatting in the println! macro):

```
pub fn App() -> Element {
    let title = "title";
    let by = "author";
    let score = 0;
    let time = chrono::Utc::now();
    let comments = "comments";

    rsx! {"{title} by {by} ({score}) {time} {comments}"}
}
```

Copy

title by author (0) 2024-08-15 01:37:52.015286087 UTC comments

Creating Elements

Next, let's wrap our post description in a `div`. You can create HTML elements in Dioxus by putting a `{` after the element name and a `}` after the last child of the element:

```
pub fn App() -> Element {
    let title = "title";
    let by = "author";
    let score = 0;
    let time = chrono::Utc::now();
    let comments = "comments";

    rsx! { div { "{title} by {by} ({score}) {time} {comments}" } }
}
```

Copy

title by author (0) 2024-08-15 01:37:52.015290375 UTC comments

> You can read more about elements in the rsx reference.

Setting Attributes

Next, let's add some padding around our post listing with an attribute.

Attributes (and listeners) modify the behavior or appearance of the element they are attached to. They are specified inside the `{}` brackets before any children, using the `name: value` syntax. You can format the text in the attribute as you would with a text node:

```
pub fn App() -> Element {
    let title = "title";
    let by = "author";
    let score = 0;
    let time = chrono::Utc::now();
    let comments = "comments";

    rsx! {
        div { padding: "0.5rem", position: "relative",
            "{title} by {by} ({score}) {time} {comments}"
        }
    }
}
```

Copy

title by author (0) 2024-08-15 01:37:52.015293500 UTC comments

> Note: All attributes defined in `dioxus-html` follow the snake_case naming convention. They transform their `snake_case` names to HTML's `camelCase` attributes.

> Note: Styles can be used directly outside of the `style:` attribute. In the above example, `padding: "0.5rem"` is turned into `style="padding: 0.5rem"`.

> You can read more about elements in the attribute reference

Creating a Component

Just like you wouldn't want to write a complex program in a single, long, `main` function, you shouldn't build a complex UI in a single `App` function. Instead, you should break down the functionality of an app in logical parts called components.

A component is a Rust function, named in UpperCamelCase, that takes a props parameter and returns an `Element` describing the UI it wants to render. In fact, our `App` function is a component!

Let's pull our story description into a new component:

```
fn StoryListing() -> Element {
    let title = "title";
    let by = "author";
    let score = 0;
    let time = chrono::Utc::now();
    let comments = "comments";

    rsx! {
        div { padding: "0.5rem", position: "relative",
            "{title} by {by} ({score}) {time} {comments}"
        }
    }
}
```

Copy

We can render our component like we would an element by putting `{}`s after the component name. Let's modify our `App` component to render our new StoryListing component:

```
pub fn App() -> Element {
    rsx! { StoryListing {} }
}
```

Copy

title by author (0) 2024-08-15 01:37:52.015296897 UTC comments

> You can read more about elements in the component reference

Creating Props

Just like you can pass arguments to a function or attributes to an element, you can pass props to a component that customize its behavior!

We can define arguments that components can take when they are rendered (called `Props`) by adding the `#[component]` macro before our function definition and adding extra function arguments.

Currently, our `StoryListing` component always renders the same story. We can modify it to accept a story to render as a prop.

We will also define what a post is and include information for how to transform our post to and from a different format using serde. This will be used with the hackernews API in a later chapter:

```
use chrono::{DateTime, Utc};
use serde::{Deserialize, Serialize};

// Define the Hackernews types
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryPageData {
    #[serde(flatten)]
    pub item: StoryItem,
    #[serde(default)]
    pub comments: Vec<Comment>,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Comment {
    pub id: i64,
    /// there will be no by field if the comment was deleted
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub text: String,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    #[serde(default)]
    pub sub_comments: Vec<Comment>,
```

Copy

```
    pub r#type: String,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryItem {
    pub id: i64,
    pub title: String,
    pub url: Option<String>,
    pub text: Option<String>,
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub score: i64,
    #[serde(default)]
    pub descendants: i64,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    pub r#type: String,
}

#[component]
fn StoryListing(story: ReadOnlySignal<StoryItem>) -> Element {
    let StoryItem {
        title,
        url,
        by,
        score,
        time,
        kids,
        ..
    } = &*story.read();

    let comments = kids.len();

    rsx! {
        div { padding: "0.5rem", position: "relative",
            "{title} by {by} ({score}) {time} {comments}"
        }
    }
}
```

Make sure to also add serde as a dependency:

```
cargo add serde --features derive
cargo add serde_json
```

Copy

We will also use the chrono crate to provide utilities for handling time data from the hackernews API:

```
cargo add chrono --features serde
```

Copy

Now, let's modify the `App` component to pass the story to our `StoryListing` component like we would set an attribute on an element:

```
pub fn App() -> Element {
    rsx! {
        StoryListing {
            story: StoryItem {
                id: 0,
                title: "hello hackernews".to_string(),
                url: None,
                text: None,
                by: "Author".to_string(),
                score: 0,
                descendants: 0,
                time: chrono::Utc::now(),
                kids: vec![],
                r#type: "".to_string(),
            }
        }
    }
}
```

Copy

hello hackernews by Author (0) 2024-08-15 01:37:52.015299351 UTC 0

> You can read more about Props in the Props reference

Cleaning Up Our Interface

Finally, by combining elements and attributes, we can make our post listing much more appealing:

Full code up to this point:

Copy

```rust
use dioxus::prelude::*;

// Define the Hackernews types
use chrono::{DateTime, Utc};
use serde::{Deserialize, Serialize};

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryPageData {
    #[serde(flatten)]
    pub item: StoryItem,
    #[serde(default)]
    pub comments: Vec<Comment>,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Comment {
    pub id: i64,
    /// there will be no by field if the comment was deleted
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub text: String,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    #[serde(default)]
    pub sub_comments: Vec<Comment>,
    pub r#type: String,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryItem {
    pub id: i64,
    pub title: String,
    pub url: Option<String>,
    pub text: Option<String>,
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub score: i64,
    #[serde(default)]
    pub descendants: i64,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    pub r#type: String,
}

fn main() {
    launch(App);
}

pub fn App() -> Element {
    rsx! {
        StoryListing {
            story: StoryItem {
                id: 0,
                title: "hello hackernews".to_string(),
                url: None,
                text: None,
                by: "Author".to_string(),
                score: 0,
                descendants: 0,
                time: Utc::now(),
                kids: vec![],
                r#type: "".to_string(),
            }
        }
    }
}

#[component]
fn StoryListing(story: ReadOnlySignal<StoryItem>) -> Element {
    let StoryItem {
        title,
        url,
        by,
        score,
        time,
        kids,
        ..
    } = &*story.read();
```

```
    let url = url.as_deref().unwrap_or_default();
    let hostname = url
        .trim_start_matches("https://")
        .trim_start_matches("http://")
        .trim_start_matches("www.");
    let score = format!("{score} {}", if *score == 1 { " point" } else { " po
    let comments = format!(
        "{} {}",
        kids.len(),
        if kids.len() == 1 {
            " comment"
        } else {
            " comments"
        }
    );
    let time = time.format("%D %l:%M %p");

    rsx! {
        div { padding: "0.5rem", position: "relative",
            div { font_size: "1.5rem",
                a { href: url, "{title}" }
                a {
                    color: "gray",
                    href: "https://news.ycombinator.com/from?site={hostname}"
                    text_decoration: "none",
                    " ({hostname})"
                }
            }
            div { display: "flex", flex_direction: "row", color: "gray",
                div { "{score}" }
                div { padding_left: "0.5rem", "by {by}" }
                div { padding_left: "0.5rem", "{time}" }
                div { padding_left: "0.5rem", "{comments}" }
            }
        }
    }
}
```

hello hackernews ()

0 points by Author 08/15/24 1:37 AM 0 comments

Interactivity

In this chapter, we will add a preview for articles you hover over or links you focus on.

Creating a Preview

First, let's split our app into a Stories component on the left side of the screen, and a preview component on the right side of the screen:

```
pub fn App() -> Element {
    rsx! {
        div { display: "flex", flex_direction: "row", width: "100%",
            div { width: "50%", Stories {} }
            div { width: "50%", Preview {} }
        }
    }
}

// New
fn Stories() -> Element {
    rsx! {
        StoryListing {
            story: StoryItem {
                id: 0,
                title: "hello hackernews".to_string(),
                url: None,
                text: None,
                by: "Author".to_string(),
                score: 0,
                descendants: 0,
                time: chrono::Utc::now(),
                kids: vec![],
                r#type: "".to_string(),
            }
        }
    }
}

// New
#[derive(Clone, Debug)]
enum PreviewState {
    Unset,
    Loading,
    Loaded(StoryPageData),
}

// New
fn Preview() -> Element {
    let preview_state = PreviewState::Unset;
    match preview_state {
        PreviewState::Unset => rsx! {"Hover over a story to preview it here"}
        PreviewState::Loading => rsx! {"Loading..."},
        PreviewState::Loaded(story) => {
            rsx! {
                div { padding: "0.5rem",
                    div { font_size: "1.5rem", a { href: story.item.url, "{st
                    div { dangerous_inner_html: story.item.text }
                    for comment in &story.comments {
                        Comment { comment: comment.clone() }
                    }
                }
            }
        }
    }
}

// NEW
#[component]
fn Comment(comment: Comment) -> Element {
    rsx! {
        div { padding: "0.5rem",
            div { color: "gray", "by {comment.by}" }
            div { dangerous_inner_html: "{comment.text}" }
            for kid in &comment.sub_comments {
                Comment { comment: kid.clone() }
            }
        }
    }
}
```

Copy

Edit this page!

Go to version
0.5
0.4
0.3

hello hackernews ()

0 points by Author 08/15/24 1:37 AM 0 comments

Hover over a story to preview it here

Event Handlers

Next, we need to detect when the user hovers over a section or focuses a link. We can use an event listener to listen for the hover and focus events.

Event handlers are similar to regular attributes, but their name usually starts with `on` - and they accept closures as values. The closure will be called whenever the event it listens for is triggered. When an event is triggered, information about the event is passed to the closure through the Event structure.

Let's create a `onmouseenter` event listener in the `StoryListing` component:

```
rsx! {
    div {
        padding: "0.5rem",
        position: "relative",
        onmouseenter: move |_| {},
        div { font_size: "1.5rem",
            a { href: url, onfocus: move |_event| {}, "{title}" }
            a {
                color: "gray",
                href: "https://news.ycombinator.com/from?site={hostname}",
                text_decoration: "none",
                " ({hostname})"
            }
        }
        div { display: "flex", flex_direction: "row", color: "gray",
            div { "{score}" }
            div { padding_left: "0.5rem", "by {by}" }
            div { padding_left: "0.5rem", "{time}" }
            div { padding_left: "0.5rem", "{comments}" }
        }
    }
}
```

> You can read more about Event Handlers in the Event Handler reference

State

So far our components have had no state like normal rust functions. To make our application change when we hover over a link we need state to store the currently hovered link in the root of the application.

You can create state in dioxus using hooks. Hooks are Rust functions you call in a constant order in a component that add additional functionality to the component.

In this case, we will use the `use_context_provider` and `use_context` hooks:

- You can provide a closure to `use_context_provider` that determines the initial value of the shared state and provides the value to all child components
- You can then use the `use_context` hook to read and modify that state in the `Preview` and `StoryListing` components
- When the value updates, the `Signal` will cause the component to re-render, and provides you with the new value

> Note: You should prefer local state hooks like use_signal or use_signal_sync when you only use state in one component. Because we use state in multiple components, we can use a global state pattern

```
pub fn App() -> Element {
    use_context_provider(|| Signal::new(PreviewState::Unset));
```

```
#[component]
fn StoryListing(story: ReadOnlySignal<StoryItem>) -> Element {
    let mut preview_state = consume_context::<Signal<PreviewState>>();
    let StoryItem {
        title,
        url,
        by,
        score,
        time,
        kids,
        ..
```

```
    } = &*story.read();

    let url = url.as_deref().unwrap_or_default();
    let hostname = url
        .trim_start_matches("https://")
        .trim_start_matches("http://")
        .trim_start_matches("www.");
    let score = format!("{score} point{}", if *score > 1 { "s" } else { "" }
    let comments = format!(
        "{} {}",
        kids.len(),
        if kids.len() == 1 {
            " comment"
        } else {
            " comments"
        }
    );
    let time = time.format("%D %l:%M %p");

    rsx! {
        div {
            padding: "0.5rem",
            position: "relative",
            onmouseenter: move |_event| {
                *preview_state
                    .write() = PreviewState::Loaded(StoryPageData {
                    item: story(),
                    comments: vec![],
                });
            },
            div { font_size: "1.5rem",
                a {
                    href: url,
                    onfocus: move |_event| {
                        *preview_state
                            .write() = PreviewState::Loaded(StoryPageData {
                            item: story(),
                            comments: vec![],
                        });
                    },
```

Copy

```
fn Preview() -> Element {
    // New
    let preview_state = consume_context::<Signal<PreviewState>>();

    // New
    match preview_state() {
```

hello hackernews ()

0 point by Author 08/15/24 1:37 AM 0 comments

Hover over a story to preview it here

> You can read more about Hooks in the Hooks reference

The Rules of Hooks

Hooks are a powerful way to manage state in Dioxus, but there are some rules you need to follow to insure they work as expected. Dioxus uses the order you call hooks to differentiate between hooks. Because the order you call hooks matters, you must follow these rules:

1. Hooks may be only used in components or other hooks (we'll get to that later)
2. On every call to the component function
    i. The same hooks must be called
    ii. In the same order
3. Hooks name's should start with `use_` so you don't accidentally confuse them with regular functions

These rules mean that there are certain things you can't do with hooks:

No Hooks in Conditionals

```
// ❌ don't call hooks in conditionals!
// We must ensure that the same hooks will be called every time
// But `if` statements only run if the conditional is true!
// So we might violate rule 2.
if you_are_happy && you_know_it {
    let something = use_signal(|| "hands");
    println!("clap your {something}")
}

// ✅ instead, *always* call use_signal
```

```rust
// You can put other stuff in the conditional though
let something = use_signal(|| "hands");
if you_are_happy && you_know_it {
    println!("clap your {something}")
}
```

No Hooks in Closures

```rust
// ❌ don't call hooks inside closures!
// We can't guarantee that the closure, if used, will be called in the same o
let _a = || {
    let b = use_signal(|| 0);
    b()
};

// ✅ instead, move hook `b` outside
let b = use_signal(|| 0);
let _a = || b();
```

No Hooks in Loops

```rust
// `names` is a Vec<&str>

// ❌ Do not use hooks in loops!
// In this case, if the length of the Vec changes, we break rule 2
for _name in &names {
    let is_selected = use_signal(|| false);
    println!("selected: {is_selected}");
}

// ✅ Instead, use a hashmap with use_signal
let selection_map = use_signal(HashMap::<&str, bool>::new);

for name in &names {
    let is_selected = selection_map.read()[name];
    println!("selected: {is_selected}");
}
```

Fetching Data

In this chapter, we will fetch data from the hacker news API and use it to render the list of top posts in our application.

Defining the API

First we need to create some utilities to fetch data from the hackernews API using reqwest:

```rust
// Define the Hackernews API
use futures::future::join_all;

pub static BASE_API_URL: &str = "https://hacker-news.firebaseio.com/v0/";
pub static ITEM_API: &str = "item/";
pub static USER_API: &str = "user/";
const COMMENT_DEPTH: i64 = 2;

pub async fn get_story_preview(id: i64) -> Result<StoryItem, reqwest::Error>
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    reqwest::get(&url).await?.json().await
}

pub async fn get_stories(count: usize) -> Result<Vec<StoryItem>, reqwest::Err
    let url = format!("{}topstories.json", BASE_API_URL);
    let stories_ids = &reqwest::get(&url).await?.json::<Vec<i64>>().await?[..

    let story_futures = stories_ids[..usize::min(stories_ids.len(), count)]
        .iter()
        .map(|&story_id| get_story_preview(story_id));
    let stories = join_all(story_futures)
        .await
        .into_iter()
        .filter_map(|story| story.ok())
        .collect();
    Ok(stories)
}

pub async fn get_story(id: i64) -> Result<StoryPageData, reqwest::Error> {
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    let mut story = reqwest::get(&url).await?.json::<StoryPageData>().await?;
    let comment_futures = story.item.kids.iter().map(|&id| get_comment(id));
    let comments = join_all(comment_futures)
        .await
        .into_iter()
        .filter_map(|c| c.ok())
        .collect();

    story.comments = comments;
    Ok(story)
}

#[async_recursion::async_recursion(?Send)]
pub async fn get_comment_with_depth(id: i64, depth: i64) -> Result<Comment, r
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    let mut comment = reqwest::get(&url).await?.json::<Comment>().await?;
    if depth > 0 {
        let sub_comments_futures = comment
            .kids
            .iter()
            .map(|story_id| get_comment_with_depth(*story_id, depth - 1));
        comment.sub_comments = join_all(sub_comments_futures)
            .await
            .into_iter()
            .filter_map(|c| c.ok())
            .collect();
    }
    Ok(comment)
}

pub async fn get_comment(comment_id: i64) -> Result<Comment, reqwest::Error>
    let comment = get_comment_with_depth(comment_id, COMMENT_DEPTH).await?;
    Ok(comment)
}
```

The code above requires you to add the reqwest, async_recursion, and futures crate:

```
cargo add reqwest --features json
cargo add async_recursion
cargo add futures
```

A quick overview of the supporting crates:

- reqwest allows us to create HTTP calls to the hackernews API.
- async_recursion provides a utility macro to allow us to recursively use an async function.
- futures provides us with utilities all around Rust's futures.

Working with Async

`use_resource` is a hook that lets you run an async closure, and provides you with its result.

For example, we can make an API request (using reqwest) inside `use_resource`:

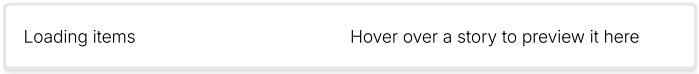
```
fn Stories() -> Element {
    // Fetch the top 10 stories on Hackernews
    let stories = use_resource(move || get_stories(10));

    // check if the future is resolved
    match &*stories.read_unchecked() {
        Some(Ok(list)) => {
            // if it is, render the stories
            rsx! {
                div {
                    // iterate over the stories with a for loop
                    for story in list {
                        // render every story with the StoryListing componen
                        StoryListing { story: story.clone() }
                    }
                }
            }
        }
        Some(Err(err)) => {
            // if there was an error, render the error
            rsx! {"An error occurred while fetching stories {err}"}
        }
        None => {
            // if the future is not resolved yet, render a loading message
            rsx! {"Loading items"}
        }
    }
}
```

The code inside `use_resource` will be submitted to the Dioxus scheduler once the component has rendered.

We can use `.read()` to get the result of the future. On the first run, since there's no data ready when the component loads, its value will be `None`. However, once the future is finished, the component will be re-rendered and the value will now be `Some(...)`, containing the return value of the closure.

We can then render the result by looping over each of the posts and rendering them with the `StoryListing` component.



> You can read more about working with Async in Dioxus in the Async reference

Lazily Fetching Data

Finally, we will lazily fetch the comments on each post as the user hovers over the post.

We need to revisit the code that handles hovering over an item. Instead of passing an empty list of comments, we can fetch all the related comments when the user hovers over the item.

We will cache the list of comments with a use_signal hook. This hook allows you to store some state in a single component. When the user triggers fetching the comments we will check if the response has already been cached before fetching the data from the hackernews API.

```
// New
async fn resolve_story(
    mut full_story: Signal<Option<StoryPageData>>,
    mut preview_state: Signal<PreviewState>,
    story_id: i64,
) {
    if let Some(cached) = full_story.as_ref() {
        *preview_state.write() = PreviewState::Loaded(cached.clone());
```

```
        return;
    }

    *preview_state.write() = PreviewState::Loading;
    if let Ok(story) = get_story(story_id).await {
        *preview_state.write() = PreviewState::Loaded(story.clone());
        *full_story.write() = Some(story);
    }
}

#[component]
fn StoryListing(story: ReadOnlySignal<StoryItem>) -> Element {
    let mut preview_state = consume_context::<Signal<PreviewState>>();
    let StoryItem {
        title,
        url,
        by,
        score,
        time,
        kids,
        id,
        ..
    } = story();
    // New
    let full_story = use_signal(|| None);

    let url = url.as_deref().unwrap_or_default();
    let hostname = url
        .trim_start_matches("https://")
        .trim_start_matches("http://")
        .trim_start_matches("www.");
    let score = format!("{score} {}", if score == 1 { " point" } else { " poi
    let comments = format!(
        "{} {}",
        kids.len(),
        if kids.len() == 1 {
            " comment"
        } else {
            " comments"
        }
    );
    let time = time.format("%D %l:%M %p");

    rsx! {
        div {
            padding: "0.5rem",
            position: "relative",
            onmouseenter: move |_event| { resolve_story(full_story, preview_s
            div { font_size: "1.5rem",
                a {
                    href: url,
                    onfocus: move |_event| { resolve_story(full_story, previe
                    // ...
```

Loading items

Hover over a story to preview it here

Conclusion

Well done! You've completed the Dioxus guide and built a hackernews application in Dioxus.

To continue your journey, you can attempt a challenge listed below, or look at the Dioxus reference.

Challenges

- Organize your components into separate files for better maintainability.
- Give your app some style if you haven't already.
- Integrate your application with the Dioxus router.

The full code for the hacker news project

Copy

```
#![allow(non_snake_case)]
use dioxus::prelude::*;

fn main() {
    launch(App);
}

pub fn App() -> Element {
    use_context_provider(|| Signal::new(PreviewState::Unset));

    rsx! {
        div { display: "flex", flex_direction: "row", width: "100%",
            div { width: "50%", Stories {} }
            div { width: "50%", Preview {} }
        }
    }
}

fn Stories() -> Element {
    let stories = use_resource(move || get_stories(10));

    match &*stories.read_unchecked() {
        Some(Ok(list)) => rsx! {
            div {
                for story in list {
                    StoryListing { story: story.clone() }
                }
            }
        },
        Some(Err(err)) => rsx! {"An error occurred while fetching stories {e
        None => rsx! {"Loading items"},
    }
}

async fn resolve_story(
    mut full_story: Signal<Option<StoryPageData>>,
    mut preview_state: Signal<PreviewState>,
    story_id: i64,
) {
    if let Some(cached) = full_story.as_ref() {
        *preview_state.write() = PreviewState::Loaded(cached.clone());
        return;
    }

    *preview_state.write() = PreviewState::Loading;
    if let Ok(story) = get_story(story_id).await {
        *preview_state.write() = PreviewState::Loaded(story.clone());
        *full_story.write() = Some(story);
    }
}

#[component]
fn StoryListing(story: ReadOnlySignal<StoryItem>) -> Element {
    let preview_state = consume_context::<Signal<PreviewState>>();
    let StoryItem {
        title,
        url,
        by,
        score,
        time,
        kids,
        id,
        ..
    } = story();
```


Edit this page!

Go to version
0.5
0.4
0.3

```
    let full_story = use_signal(|| None);

    let url = url.as_deref().unwrap_or_default();
    let hostname = url
        .trim_start_matches("https://")
        .trim_start_matches("http://")
        .trim_start_matches("www.");
    let score = format!("{score} {}", if score == 1 { " point" } else { " poi
    let comments = format!(
        "{} {}",
        kids.len(),
        if kids.len() == 1 {
            " comment"
        } else {
            " comments"
        }
    );
    let time = time.format("%D %l:%M %p");

    rsx! {
        div {
            padding: "0.5rem",
            position: "relative",
            onmouseenter: move |_event| { resolve_story(full_story, preview_s
            div { font_size: "1.5rem",
                a {
                    href: url,
                    onfocus: move |_event| { resolve_story(full_story, previe
                    "{title}"
                }
                a {
                    color: "gray",
                    href: "https://news.ycombinator.com/from?site={hostname}"
                    text_decoration: "none",
                    " ({hostname})"
                }
            }
            div { display: "flex", flex_direction: "row", color: "gray",
                div { "{score}" }
                div { padding_left: "0.5rem", "by {by}" }
                div { padding_left: "0.5rem", "{time}" }
                div { padding_left: "0.5rem", "{comments}" }
            }
        }
    }
}

#[derive(Clone, Debug)]
enum PreviewState {
    Unset,
    Loading,
    Loaded(StoryPageData),
}

fn Preview() -> Element {
    let preview_state = consume_context::<Signal<PreviewState>>();

    match preview_state() {
        PreviewState::Unset => rsx! {"Hover over a story to preview it here"}
        PreviewState::Loading => rsx! {"Loading..."},
        PreviewState::Loaded(story) => {
            rsx! {
                div { padding: "0.5rem",
                    div { font_size: "1.5rem", a { href: story.item.url, "{st
                    div { dangerous_inner_html: story.item.text }
                    for comment in &story.comments {
                        Comment { comment: comment.clone() }
                    }
                }
            }
        }
    }
}

#[component]
fn Comment(comment: Comment) -> Element {
    rsx! {
        div { padding: "0.5rem",
            div { color: "gray", "by {comment.by}" }
            div { dangerous_inner_html: "{comment.text}" }
            for kid in &comment.sub_comments {
                Comment { comment: kid.clone() }
            }
        }
    }
}

// Define the Hackernews API and types
use chrono::{DateTime, Utc};
use futures::future::join_all;
use serde::{Deserialize, Serialize};
```

```rust
pub static BASE_API_URL: &str = "https://hacker-news.firebaseio.com/v0/";
pub static ITEM_API: &str = "item/";
pub static USER_API: &str = "user/";
const COMMENT_DEPTH: i64 = 2;

pub async fn get_story_preview(id: i64) -> Result<StoryItem, reqwest::Error>
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    reqwest::get(&url).await?.json().await
}

pub async fn get_stories(count: usize) -> Result<Vec<StoryItem>, reqwest::Err
    let url = format!("{}topstories.json", BASE_API_URL);
    let stories_ids = &reqwest::get(&url).await?.json::<Vec<i64>>().await?[..

    let story_futures = stories_ids[..usize::min(stories_ids.len(), count)]
        .iter()
        .map(|&story_id| get_story_preview(story_id));
    Ok(join_all(story_futures)
        .await
        .into_iter()
        .filter_map(|story| story.ok())
        .collect())
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryPageData {
    #[serde(flatten)]
    pub item: StoryItem,
    #[serde(default)]
    pub comments: Vec<Comment>,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Comment {
    pub id: i64,
    /// there will be no by field if the comment was deleted
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub text: String,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    #[serde(default)]
    pub sub_comments: Vec<Comment>,
    pub r#type: String,
}

#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct StoryItem {
    pub id: i64,
    pub title: String,
    pub url: Option<String>,
    pub text: Option<String>,
    #[serde(default)]
    pub by: String,
    #[serde(default)]
    pub score: i64,
    #[serde(default)]
    pub descendants: i64,
    #[serde(with = "chrono::serde::ts_seconds")]
    pub time: DateTime<Utc>,
    #[serde(default)]
    pub kids: Vec<i64>,
    pub r#type: String,
}

pub async fn get_story(id: i64) -> Result<StoryPageData, reqwest::Error> {
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    let mut story = reqwest::get(&url).await?.json::<StoryPageData>().await?
    let comment_futures = story.item.kids.iter().map(|&id| get_comment(id));
    let comments = join_all(comment_futures)
        .await
        .into_iter()
        .filter_map(|c| c.ok())
        .collect();

    story.comments = comments;
    Ok(story)
}

#[async_recursion::async_recursion(?Send)]
pub async fn get_comment_with_depth(id: i64, depth: i64) -> Result<Comment, r
    let url = format!("{}{}{}.json", BASE_API_URL, ITEM_API, id);
    let mut comment = reqwest::get(&url).await?.json::<Comment>().await?;
    if depth > 0 {
        let sub_comments_futures = comment
            .kids
            .iter()
```

```
            .map(|story_id| get_comment_with_depth(*story_id, depth - 1));
        comment.sub_comments = join_all(sub_comments_futures)
            .await
            .into_iter()
            .filter_map(|c| c.ok())
            .collect();
    }
    Ok(comment)
}

pub async fn get_comment(comment_id: i64) -> Result<Comment, reqwest::Error>
    get_comment_with_depth(comment_id, COMMENT_DEPTH).await
}
```

Describing the UI

Dioxus is a *declarative* framework. This means that instead of telling Dioxus what to do (e.g. to "create an element" or "set the color to red") we simply *declare* what we want the UI to look like using RSX.

You have already seen a simple example of RSX syntax in the "hello world" application:

```
// define a component that renders a div with the text "Hello, world!"
fn App() -> Element {
    rsx! { div { "Hello, world!" } }
}
```

Here, we use the `rsx!` macro to *declare* that we want a `div` element, containing the text `"Hello, world!"`. Dioxus takes the RSX and constructs a UI from it.

RSX Features

RSX is very similar to HTML in that it describes elements with attributes and children. Here's an empty `button` element in RSX, as well as the resulting HTML:

```
rsx! {
    button {
        // attributes / listeners
        // children
        "Hello, World!"
    }
}
```

Hello, World!

Attributes

Attributes (and event handlers) modify the behavior or appearance of the element they are attached to. They are specified inside the `{}` brackets, using the `name: value` syntax. You can provide the value as a literal in the RSX:

```
rsx! {
    img {
        src: "https://avatars.githubusercontent.com/u/79236386?s=200&v=4",
        class: "primary_button",
        width: "10px"
    }
}
```

Some attributes, such as the `type` attribute for `input` elements won't work on their own in Rust. This is because `type` is a reserved Rust keyword. To get around this, Dioxus uses the `r#` specifier:

```
rsx! { input { r#type: "text", color: "red" } }
```

> Note: All attributes defined in `dioxus-html` follow the snake_case naming convention. They transform their `snake_case` names to HTML's `camelCase` attributes.

> Note: Styles can be used directly outside of the `style:` attribute. In the above example, `color: "red"` is turned into `style="color: red"`.

Conditional Attributes

You can also conditionally include attributes by using an if statement without an else branch. This is useful for adding an attribute only if a certain condition is met:

```
let large_font = true;
rsx! { div { class: if large_font { "text-xl" }, "Hello, World!" } }
```


Edit this page!

Go to version
0.5
0.4
0.3

Hello, World!

Custom Attributes

Dioxus has a pre-configured set of attributes that you can use. RSX is validated at compile time to make sure you didn't specify an invalid attribute. If you want to override this behavior with a custom attribute name, specify the attribute in quotes:

```
rsx! { div { "style": "width: 20px; height: 20px; background-color: re
```
Copy

Special Attributes

While most attributes are simply passed on to the HTML, some have special behaviors.

The HTML Escape Hatch

If you're working with pre-rendered assets, output from templates, or output from a JS library, then you might want to pass HTML directly instead of going through Dioxus. In these instances, reach for `dangerous_inner_html`.

For example, shipping a markdown-to-Dioxus converter might significantly bloat your final application size. Instead, you'll want to pre-render your markdown to HTML and then include the HTML directly in your output. We use this approach for the Dioxus homepage:

```
// this should come from a trusted source
let contents = "live <b>dangerously</b>";

rsx! { div { dangerous_inner_html: "{contents}" } }
```
Copy

live **dangerously**

> Note! This attribute is called "dangerous_inner_html" because it is **dangerous** to pass it data you don't trust. If you're not careful, you can easily expose cross-site scripting (XSS) attacks to your users.
>
> If you're handling untrusted input, make sure to sanitize your HTML before passing it into `dangerous_inner_html` – or just pass it to a Text Element to escape any HTML tags.

Boolean Attributes

Most attributes, when rendered, will be rendered exactly as the input you provided. However, some attributes are considered "boolean" attributes and just their presence determines whether they affect the output. For these attributes, a provided value of `"false"` will cause them to be removed from the target element.

So this RSX wouldn't actually render the `hidden` attribute:

```
rsx! { div { hidden: false, "hello" } }
```
Copy

hello

Not all attributes work like this however. *Only the following attributes* have this behavior:

- `allowfullscreen`
- `allowpaymentrequest`
- `async`
- `autofocus`
- `autoplay`
- `checked`
- `controls`
- `default`
- `defer`
- `disabled`
- `formnovalidate`

- `hidden`
- `ismap`
- `itemscope`
- `loop`
- `multiple`
- `muted`
- `nomodule`
- `novalidate`
- `open`
- `playsinline`
- `readonly`
- `required`
- `reversed`
- `selected`
- `truespeed`

For any other attributes, a value of `"false"` will be sent directly to the DOM.

Interpolation

Similarly to how you can format Rust strings, you can also interpolate in RSX text. Use `{variable}` to Display the value of a variable in a string, or `{variable:?}` to use the Debug representation:

```
let coordinates = (42, 0);
let country = "es";
rsx! {
    div {
        class: "country-{country}",
        left: "{coordinates.0:?}",
        top: "{coordinates.1:?}",
        // arbitrary expressions are allowed,
        // as long as they don't contain `{}`
        div { "{country.to_uppercase()}" }
        div { "{7*6}" }
        // {} can be escaped with {{}}
        div { "{{}}" }
    }
}
```

ES
42
{}

Children

To add children to an element, put them inside the `{}` brackets after all attributes and listeners in the element. They can be other elements, text, or components. For example, you could have an `ol` (ordered list) element, containing 3 `li` (list item) elements, each of which contains some text:

```
rsx! {
    ol {
        li { "First Item" }
        li { "Second Item" }
        li { "Third Item" }
    }
}
```

1. First Item
2. Second Item
3. Third Item

Fragments

You can render multiple elements at the top level of `rsx!` and they will be automatically grouped.

```
rsx! {
    p { "First Item" }
```

```
    p { "Second Item" }
}
```

First Item

Second Item

Expressions

You can include arbitrary Rust expressions as children within RSX by surrounding your expression with `{}`s. Any expression that implements IntoDynNode can be used within rsx. This is useful for displaying data from an iterator:

```
let text = "Dioxus";
rsx! {
    span {
        {text.to_uppercase()},
        // create a list of text from 0 to 9
        {(0..10).map(|i| rsx!{ "{i}" })}
    }
}
```

DIOXUS0123456789

Loops

In addition to iterators you can also use for loops directly within RSX:

```
rsx! {
    // use a for loop where the body itself is RSX
    div {
        // create a list of text from 0 to 9
        for i in 0..3 {
            // NOTE: the body of the loop is RSX not a rust statement
            div { "{i}" }
        }
    }
    // iterator equivalent
    div { {(0..3).map(|i| rsx!{ div { "{i}" } })} }
}
```

0
1
2
0
1
2

If statements

You can also use if statements without an else branch within RSX:

```
rsx! {
    // use if statements without an else
    if true {
        div { "true" }
    }
}
```

true

An Open Source project dedicated to making Rust UI wonderful.

Twitter
Discord

Guide
Awesome

CLI
Taffy

Components

Just like you wouldn't want to write a complex program in a single, long, `main` function, you shouldn't build a complex UI in a single `App` function. Instead, you should break down the functionality of an app in logical parts called components.

A component is a Rust function, named in UpperCammelCase, that either takes no parameters or a properties struct and returns an `Element` describing the UI it wants to render.

```
// define a component that renders a div with the text "Hello, world!"
fn App() -> Element {
    rsx! { div { "Hello, world!" } }
}
```

> You'll probably want to add `#![allow(non_snake_case)]` to the top of your crate to avoid warnings about UpperCammelCase component names

A Component is responsible for some rendering task – typically, rendering an isolated part of the user interface. For example, you could have an `About` component that renders a short description of Dioxus Labs:

```
pub fn About() -> Element {
    rsx! {
        p {
            b { "Dioxus Labs" }
            " An Open Source project dedicated to making Rust UI wonderful."
        }
    }
}
```

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

Then, you can render your component in another component, similarly to how elements are rendered:

```
pub fn App() -> Element {
    rsx! {
        About {}
        About {}
    }
}
```

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

> At this point, it might seem like components are nothing more than functions. However, as you learn more about the features of Dioxus, you'll see that they are actually more powerful!

Component Props

Just like you can pass arguments to a function or attributes to an element, you can pass props to a component that customize its behavior! The components we've seen so far didn't accept any props – so let's write some components that do.

`#[derive(Props)]`

Component props are a single struct annotated with `#[derive(PartialEq, Clone, Props)]`. For a component to accept props, the type of its argument must be `YourPropsStruct`.

Example:

```
// Remember: Owned props must implement `PartialEq`!
#[derive(PartialEq, Props, Clone)]
struct LikesProps {
    score: i32,
}

fn Likes(props: LikesProps) -> Element {
    rsx! {
        div {
            "This post has "
            b { "{props.score}" }
            " likes"
        }
    }
}
```

Copy

You can then pass prop values to the component the same way you would pass attributes to an element:

```
pub fn App() -> Element {
    rsx! { Likes { score: 42 } }
}
```

Copy

This post has **42** likes

Prop Options

The `#[derive(Props)]` macro has some features that let you customize the behavior of props.

Optional Props

You can create optional fields by using the `Option<…>` type for a field:

```
#[derive(PartialEq, Clone, Props)]
struct OptionalProps {
    title: String,
    subtitle: Option<String>,
}

fn Title(props: OptionalProps) -> Element {
    rsx! {
        h1 { "{props.title}: ", {props.subtitle.unwrap_or_else(|| "No subtit
    }
}
```

Copy

Then, you can choose to either provide them or not:

```
Title { title: "Some Title" }
Title { title: "Some Title", subtitle: "Some Subtitle" }
// Providing an Option explicitly won't compile though:
// Title {
//     title: "Some Title",
//     subtitle: None,
// },
```

Copy

Explicitly Required Option

Edit this page!

Go to version
0.5
0.4
0.3

If you want to explicitly require an `Option`, and not an optional prop, you can annotate it with `#[props(!optional)]`:

```
#[derive(PartialEq, Clone, Props)]
struct ExplicitOptionProps {
    title: String,
    #[props(!optional)]
    subtitle: Option<String>,
}

fn ExplicitOption(props: ExplicitOptionProps) -> Element {
    rsx! {
        h1 { "{props.title}: ", {props.subtitle.unwrap_or_else(|| "No subtit
    }
}
```

Then, you have to explicitly pass either `Some("str")` or `None`:

```
ExplicitOption { title: "Some Title", subtitle: None }
ExplicitOption { title: "Some Title", subtitle: Some("Some Title".to_string(
// This won't compile:
// ExplicitOption {
//     title: "Some Title",
// },
```

Default Props

You can use `#[props(default = 42)]` to make a field optional and specify its default value:

```
#[derive(PartialEq, Props, Clone)]
struct DefaultProps {
    // default to 42 when not provided
    #[props(default = 42)]
    number: i64,
}

fn DefaultComponent(props: DefaultProps) -> Element {
    rsx! { h1 { "{props.number}" } }
}
```

Then, similarly to optional props, you don't have to provide it:

```
DefaultComponent { number: 5 }
DefaultComponent {}
```

Automatic Conversion with into

It is common for Rust functions to accept `impl Into<SomeType>` rather than just `SomeType` to support a wider range of parameters. If you want similar functionality with props, you can use `#[props(into)]`. For example, you could add it on a `String` prop – and `&str` will also be automatically accepted, as it can be converted into `String`:

```
#[derive(PartialEq, Props, Clone)]
struct IntoProps {
    #[props(into)]
    string: String,
}

fn IntoComponent(props: IntoProps) -> Element {
    rsx! { h1 { "{props.string}" } }
}
```

Then, you can use it so:

```
IntoComponent { string: "some &str" }
```

The component macro

So far, every Component function we've seen had a corresponding ComponentProps struct to pass in props. This was quite verbose... Wouldn't it be nice to have props as simple function arguments? Then we wouldn't need to define a Props struct, and instead of typing `props.whatever`, we could just use `whatever` directly!

`component` allows you to do just that. Instead of typing the "full" version:

```rust
#[derive(Props, Clone, PartialEq)]
struct TitleCardProps {
    title: String,
}

fn TitleCard(props: TitleCardProps) -> Element {
    rsx!{
        h1 { "{props.title}" }
    }
}
```

...you can define a function that accepts props as arguments. Then, just annotate it with `#[component]`, and the macro will turn it into a regular Component for you:

```rust
#[component]
fn TitleCard(title: String) -> Element {
    rsx!{
        h1 { "{title}" }
    }
}
```

> While the new Component is shorter and easier to read, this macro should not be used by library authors since you have less control over Prop documentation.

Component Children

In some cases, you may wish to create a component that acts as a container for some other content, without the component needing to know what that content is. To achieve this, create a prop of type `Element`:

```rust
#[derive(PartialEq, Clone, Props)]
struct ClickableProps {
    href: String,
    body: Element,
}

fn Clickable(props: ClickableProps) -> Element {
    rsx! {
        a { href: "{props.href}", class: "fancy-button", {props.body} }
    }
}
```

Then, when rendering the component, you can pass in the output of `rsx!{...}`:

```rust
rsx! {
    Clickable {
        href: "https://www.youtube.com/watch?v=C-M2hs3sXGo",
        body: rsx! {
            "How to " i { "not" } " be seen"
        }
    }
}
```

> Warning: While it may compile, do not include the same `Element` more than once in the RSX. The resulting behavior is unspecified.

The children field

Rather than passing the RSX through a regular prop, you may wish to accept children similarly to how elements can have children. The "magic" `children` prop lets you achieve this:

```rust
#[derive(PartialEq, Clone, Props)]
struct ClickableProps {
    href: String,
    children: Element,
}

fn Clickable(props: ClickableProps) -> Element {
    rsx! {
        a { href: "{props.href}", class: "fancy-button", {props.children} }
    }
}
```

This makes using the component much simpler: simply put the RSX inside the `{}` brackets – and there is no need for a `render` call or another macro!

```rust
rsx! {
    Clickable { href: "https://www.youtube.com/watch?v=C-M2hs3sXGo",
```

```
        "How to "
        i { "not" }
        " be seen"
    }
}
```

How to *not* be seen

Event Handlers

Event handlers are used to respond to user actions. For example, an event handler could be triggered when the user clicks, scrolls, moves the mouse, or types a character.

Event handlers are attached to elements. For example, we usually don't care about all the clicks that happen within an app, only those on a particular button.

Event handlers are similar to regular attributes, but their name usually starts with `on` - and they accept closures as values. The closure will be called whenever the event it listens for is triggered and will be passed that event.

For example, to handle clicks on an element, we can specify an `onclick` handler:

```
rsx! {
    button { onclick: move |event| log::info!("Clicked! Event: {event:?}"), "
}
```

Copy

click me!

The `Event` object

Event handlers receive an `Event` object containing information about the event. Different types of events contain different types of data. For example, mouse-related events contain `MouseData`, which tells you things like where the mouse was clicked and what mouse buttons were used.

In the example above, this event data was logged to the terminal:

```
Clicked! Event: UiEvent { bubble_state: Cell { value: true }, data: Mo
Clicked! Event: UiEvent { bubble_state: Cell { value: true }, data: MouseData
```

Copy

To learn what the different event types for HTML provide, read the events module docs.

Event propagation

Some events will trigger first on the element the event originated at upward. For example, a click event on a `button` inside a `div` would first trigger the button's event listener and then the div's event listener.

> For more information about event propagation see the mdn docs on event bubbling

If you want to prevent this behavior, you can call `stop_propagation()` on the event:

```
rsx! {
    div { onclick: move |_event| {},
        "outer"
        button {
            onclick: move |event| {
                event.stop_propagation();
            },
            "inner"
        }
    }
}
```

Copy

Prevent Default

Some events have a default behavior. For keyboard events, this might be entering the typed character. For mouse events, this might be selecting some text.

In some instances, might want to avoid this default behavior. For this, you can add the `prevent_default` attribute with the name of the handler whose default behavior you want to stop. This attribute can be used for multiple handlers using their name separated by spaces:

```
rsx! {
    a {
        href: "https://example.com",
        prevent_default: "onclick",
        onclick: |_| log::info!("link clicked"),
        "example.com"
    }
}
```

example.com

Any event handlers will still be called.

> Normally, in React or JavaScript, you'd call "preventDefault" on the event in the callback. Dioxus does *not* currently support this behavior. Note: this means you cannot conditionally prevent default behavior based on the data in the event.

Handler Props

Sometimes, you might want to make a component that accepts an event handler. A simple example would be a `FancyButton` component, which accepts an `onclick` handler:

```
#[derive(PartialEq, Clone, Props)]
pub struct FancyButtonProps {
    onclick: EventHandler<MouseEvent>,
}

pub fn FancyButton(props: FancyButtonProps) -> Element {
    rsx! {
        button {
            class: "fancy-button",
            onclick: move |evt| props.onclick.call(evt),
            "click me pls."
        }
    }
}
```

Then, you can use it like any other handler:

```
rsx! {
    FancyButton {
        onclick: move |event| println!("Clicked! {event:?}"),
    }
}
```

> Note: just like any other attribute, you can name the handlers anything you want! Any closure you pass in will automatically be turned into an `EventHandler`.

Async Event Handlers

Passing `EventHandler`s as props does not support passing a closure that returns an async block. Instead, you must manually call `spawn` to do async operations:

```
rsx! {
    FancyButton {
        // This does not work!
        // onclick: move |event| async move {
        //      println!("Clicked! {event:?}");
        // },

        // This does work!
        onclick: move |event| {
            spawn(async move {
                println!("Clicked! {event:?}");
            });
        },
    }
}
```

This is only the case for custom event handlers as props.

Custom Data

Event Handlers are generic over any type, so you can pass in any data you want to them, e.g:

```rust
struct ComplexData(i32);

#[derive(PartialEq, Clone, Props)]
pub struct CustomFancyButtonProps {
    onclick: EventHandler<ComplexData>,
}

pub fn CustomFancyButton(props: CustomFancyButtonProps) -> Element {
    rsx! {
        button {
            class: "fancy-button",
            onclick: move |_| props.onclick.call(ComplexData(0)),
            "click me pls."
        }
    }
}
```

Copy

Choosing a web renderer

Dioxus has three different renderers that target the web:

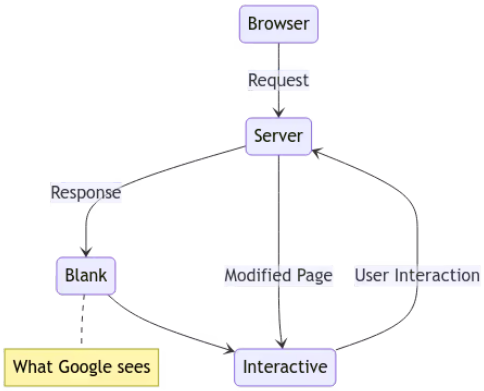
- dioxus-web allows you to render your application to HTML with WebAssembly on the client
- dioxus-liveview allows you to run your application on the server and render it to HTML on the client with a websocket
- dioxus-fullstack allows you to initially render static HTML on the server and then update that HTML from the client with WebAssembly

Each approach has its tradeoffs:

Dioxus Liveview

- Liveview rendering communicates with the server over a WebSocket connection. It essentially moves all of the work that Client-side rendering does to the server.
- This makes it **easy to communicate with the server, but more difficult to communicate with the client/browser APIS**.
- Each interaction also requires a message to be sent to the server and back which can cause **issues with latency**.
- Because Liveview uses a websocket to render, the page will be blank until the WebSocket connection has been established and the first renderer has been sent from the websocket. Just like with client side rendering, this can make the page **less SEO-friendly**.
- Because the page is rendered on the server and the page is sent to the client piece by piece, you never need to send the entire application to the client. The initial load time can be faster than client-side rendering with large applications because Liveview only needs to send a constant small websocket script regardless of the size of the application.

> Liveview is a good fit for applications that already need to communicate with the server frequently (like real time collaborative apps), but don't need to communicate with as many client/browser APIs.



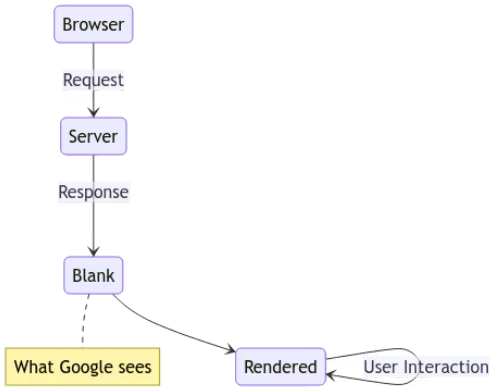
Dioxus Web

- With Client side rendering, you send your application to the client, and then the client generates all of the HTML of the page dynamically.
- This means that the page will be blank until the JavaScript bundle has loaded and the application has initialized. This can result in **slower first render times and poor SEO performance**.

> SEO stands for Search Engine Optimization. It refers to the practice of making your website more likely to appear in search engine results. Search engines like Google and Bing use web crawlers to index the content of websites. Most of these crawlers are not able to run JavaScript, so they will not be able to index the content of your page if it is rendered client-side.

- Client-side rendered applications need to use **weakly typed requests to communicate with the server**.

> Client-side rendering is a good starting point for most applications. It is well supported and makes it easy to communicate with the client/browser APIs.



Dioxus Fullstack

Fullstack rendering happens in two parts:

1. The page is rendered on the server. This can include fetching any data you need to render the page.
2. The page is hydrated on the client. (Hydration is taking the HTML page from the server and adding all of the event listeners Dioxus needs on the client). Any updates to the page happen on the client after this point.

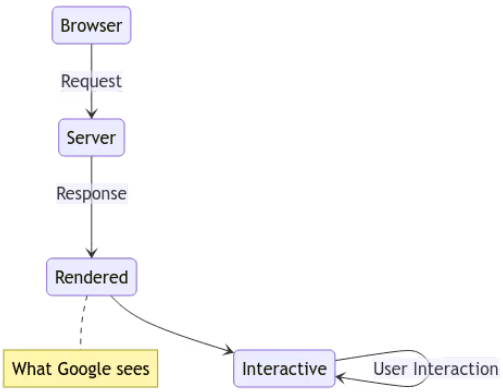
Because the page is initially rendered on the server, the page will be fully rendered when it is sent to the client. This results in a faster first render time and makes the page more SEO-friendly.

- **Fast initial render**
- **Works well with SEO**
- **Type safe easy communication with the server**
- **Access to the client/browser APIs**
- **Fast interactivity**

Finally, we can use server functions to communicate with the server in a type-safe way.

This approach uses both the dioxus-web and dioxus-ssr crates. To integrate those two packages Dioxus provides the `dioxus-fullstack` crate.

There can be more complexity with fullstack applications because your code runs in two different places. Dioxus tries to mitigate this with server functions and other helpers.



Dioxus Labs

An Open Source project dedicated to making Rust UI wonderful.

Community

Github
Twitter
Discord

Learning

docs.rs
Guide
Awesome

Projects

Dioxus
CLI
Taffy

Web

To run on the Web, your app must be compiled to WebAssembly and depend on the `dioxus` and `dioxus-web` crates.

A build of Dioxus for the web will be roughly equivalent to the size of a React build (70kb vs 65kb) but it will load significantly faster because WebAssembly can be compiled as it is streamed.

Examples:

- TodoMVC
- Tailwind App

> Note: Because of the limitations of Wasm, not every crate will work with your web apps, so you'll need to make sure that your crates work without native system calls (timers, IO, etc).

Support

The Web is the best-supported target platform for Dioxus.

- Because your app will be compiled to WASM you have access to browser APIs through wasm-bindgen.
- Dioxus provides hydration to resume apps that are rendered on the server. See the fullstack reference for more information.

Running Javascript

Dioxus provides some ergonomic wrappers over the browser API, but in some cases you may need to access parts of the browser API Dioxus does not expose.

For these cases, Dioxus web exposes the use_eval hook that allows you to run raw Javascript in the webview:

```rust
use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    // You can create as many eval instances as you want
    let mut eval = eval(
        r#"
        // You can send messages from JavaScript to Rust with the dioxus.send
        dioxus.send("Hi from JS!");
        // You can receive messages from Rust to JavaScript with the dioxus.r
        let msg = await dioxus.recv();
        console.log(msg);
        "#,
    );

    // You can send messages to JavaScript with the send method
    eval.send("Hi from Rust!".into()).unwrap();

    let future = use_resource(move || {
        to_owned![eval];
```

```
        async move {
            // You can receive any message from JavaScript with the recv meth
            eval.recv().await.unwrap()
        }
    });

    match future.read_unchecked().as_ref() {
        Some(v) => rsx! { p { "{v}" } },
        _ => rsx! { p { "hello" } },
    }
}
```

If you are targeting web, but don't plan on targeting any other Dioxus renderer you can also use the generated wrappers in the web-sys and gloo crates.

Customizing Index Template

Dioxus supports providing custom index.html templates. The index.html must include a `div` with the id `main` to be used. Hot Reload is still supported. An example is provided in the PWA-Example.

Assets

> ⚠ Support: Manganis is currently in alpha. API changes are planned and bugs are more likely

Assets are files that are included in the final build of the application. They can be images, fonts, stylesheets, or any other file that is not a source file. Dioxus includes first class support for assets, and provides a simple way to include them in your application and automatically optimize them for production.

Assets in dioxus are also compatible with libraries! If you are building a library, you can include assets in your library and they will be automatically included in the final build of any application that uses your library.

First, you need to add the `manganis` crate to your `Cargo.toml` file:

```
cargo add manganis
```
Copy

Including images

To include an asset in your application, you can simply wrap the path to the asset in a `mg!` call. For example, to include an image in your application, you can use the following code:

```
use dioxus::prelude::*;
use manganis::*;

fn App() -> Element {
    // You can link to assets that are relative to the package root or even
    // These assets will automatically be picked up by the dioxus cli, optim
    const ASSET: manganis::ImageAsset = manganis::mg!(image("./public/static/
    rsx! { img { src: "{ASSET}" } }
}
```
Copy

You can also optimize, resize, and preload images using the `mg!` macro. Choosing an optimized file type (like WebP) and a reasonable quality setting can significantly reduce the size of your images which helps your application load faster. For example, you can use the following code to include an optimized image in your application:

```
pub const ENUM_ROUTER_IMG: manganis::ImageAsset =
    manganis::mg!(image("./public/static/enum_router.png")
        // Manganis uses the builder pattern inside the macro. You can set th
        .size(52, 52)
        // You can also convert the image to a web friendly format at compile
        .format(ImageType::Avif)
        // You can even tell manganis to preload the image so it's ready to l
        .preload());

fn EnumRouter() -> Element {
    rsx! { img { src: "{ENUM_ROUTER_IMG}" } }
}
```
Copy

Including arbitrary files

In dioxus desktop, you may want to include a file with data for your application. You can use the `file` function to include arbitrary files in your application. For example, you can use the following code to include a file in your application:

```
// You can also collect arbitrary files. Relative paths are resolved r
const PATH_TO_BUNDLED_CARGO_TOML: &str = manganis::mg!(file("./Cargo.toml"))
// You can use URLs to copy the asset at build time
const PATH_TO_BUNDLED_AWESOME_DIOXUS: &str = manganis::mg!(file("https://dio
```
Copy

These files will be automatically included in the final build of your application, and you can use them in your application as you would any other file.

Including stylesheets

You can include stylesheets in your application using the `mg!` macro. For example, you can use the following code to include a stylesheet in your application:

```
// You can also bundle stylesheets with your application
// Any files that end with .css will be minified and bundled with your applic
const _: &str = manganis::mg!(file("./tailwind.css"));
```

> The tailwind guide has more information on how to use tailwind with dioxus.

Conclusion

Dioxus provides first class support for assets, and makes it easy to include them in your application. You can include images, arbitrary files, and stylesheets in your application, and dioxus will automatically optimize them for production. This makes it easy to include assets in your application and ensure that they are optimized for production.

You can read more about assets and all the options available to optimize your assets in the manganis documentation.

Spawning Futures

The `use_resource` and `use_coroutine` hooks are useful if you want to unconditionally spawn the future. Sometimes, though, you'll want to only spawn a future in response to an event, such as a mouse click. For example, suppose you need to send a request when the user clicks a "log in" button. For this, you can use `spawn`:

```
let mut response = use_signal(|| String::from("..."));

let log_in = move |_| {
    spawn(async move {
        let resp = reqwest::Client::new()
            .get("https://dioxuslabs.com")
            .send()
            .await;

        match resp {
            Ok(_data) => {
                log::info!("dioxuslabs.com responded!");
                response.set("dioxuslabs.com responded!".into());
            }
            Err(err) => {
                log::info!("Request failed with error: {err:?}")
            }
        }
    });
};

rsx! { button { onclick: log_in, "Response: {response}" } }
```

Copy

Response: ...

> Note: `spawn` will always spawn a *new* future. You most likely don't want to call it on every render.

Calling `spawn` will give you a `JoinHandle` which lets you cancel or pause the future.

Spawning Tokio Tasks

Sometimes, you might want to spawn a background task that needs multiple threads or talk to hardware that might block your app code. In these cases, we can directly spawn a Tokio task from our future. For Dioxus-Desktop, your task will be spawned onto Tokio's Multithreaded runtime:

```
spawn(async {
    let _ = tokio::spawn(async {}).await;

    let _ = tokio::task::spawn_local(async {
        // some !Send work
    })
    .await;
});
```

Copy

Coroutines

Another tool in your async toolbox are coroutines. Coroutines are futures that can have values sent to them.

Like regular futures, code in a coroutine will run until the next `await` point before yielding. This low-level control over asynchronous tasks is quite powerful, allowing for infinitely looping tasks like WebSocket polling, background timers, and other periodic actions.

`use_coroutine`

The `use_coroutine` hook allows you to create a coroutine. Most coroutines we write will be polling loops using await.

```
use futures_util::StreamExt;

fn app() {
    let ws: Coroutine<()> = use_coroutine(|rx| async move {
        // Connect to some sort of service
        let mut conn = connect_to_ws_server().await;

        // Wait for data on the service
        while let Some(msg) = conn.next().await {
            // handle messages
        }
    });
}
```

Copy

For many services, a simple async loop will handle the majority of use cases.

Yielding Values

To yield values from a coroutine, simply bring in a `Signal` handle and set the value whenever your coroutine completes its work.

The future must be `'static` – so any values captured by the task cannot carry any references to `cx`, such as a `Signal`.

You can use to_owned to create a clone of the hook handle which can be moved into the async closure.

```
let sync_status = use_signal(|| Status::Launching);
let sync_task = use_coroutine(|rx: UnboundedReceiver<SyncAction>| {
    let mut sync_status = sync_status.to_owned();
    async move {
        loop {
            tokio::time::sleep(Duration::from_secs(1)).await;
            sync_status.set(Status::Working);
        }
    }
});
```

Copy

To make this a bit less verbose, Dioxus exports the `to_owned!` macro which will create a binding as shown above, which can be quite helpful when dealing with many values.

```
let sync_status = use_signal(|| Status::Launching);
let load_status = use_signal(|| Status::Launching);
let sync_task = use_coroutine(|rx: UnboundedReceiver<SyncAction>| {
    async move {
        // ...
    }
});
```

Copy

Sending Values

You might've noticed the `use_coroutine` closure takes an argument called `rx`. What is that? Well, a common pattern in complex apps is to handle a bunch of async code at once. With libraries like Redux Toolkit, managing multiple promises at once can be challenging and a common source of bugs.

With Coroutines, we can centralize our async logic. The `rx` parameter is an Channel that allows code external to the coroutine to send data *into* the coroutine. Instead of

looping on an external service, we can loop on the channel itself, processing messages from within our app without needing to spawn a new future. To send data into the coroutine, we would call "send" on the handle.

```
use futures_util::StreamExt;

enum ProfileUpdate {
    SetUsername(String),
    SetAge(i32),
}

let profile = use_coroutine(|mut rx: UnboundedReceiver<ProfileUpdate>| async
    let mut server = connect_to_server().await;

    while let Some(msg) = rx.next().await {
        match msg {
            ProfileUpdate::SetUsername(name) => server.update_username(name).
            ProfileUpdate::SetAge(age) => server.update_age(age).await,
        }
    }
});

rsx! {
    button { onclick: move |_| profile.send(ProfileUpdate::SetUsername("Bob"
        "Update username"
    }
}
```

> Note: In order to use/run the `rx.next().await` statement you will need to extend the [`Stream`] trait (used by [`UnboundedReceiver`]) by adding 'futures_util' as a dependency to your project and adding the `use futures_util::stream::StreamExt;`.

For sufficiently complex apps, we could build a bunch of different useful "services" that loop on channels to update the app.

```
let profile = use_coroutine(profile_service);
let editor = use_coroutine(editor_service);
let sync = use_coroutine(sync_service);

async fn profile_service(rx: UnboundedReceiver<ProfileCommand>) {
    // do stuff
}

async fn sync_service(rx: UnboundedReceiver<SyncCommand>) {
    // do stuff
}

async fn editor_service(rx: UnboundedReceiver<EditorCommand>) {
    // do stuff
}
```

We can combine coroutines with Global State to emulate Redux Toolkit's Thunk system with much less headache. This lets us store all of our app's state *within* a task and then simply update the "view" values stored in Atoms. It cannot be understated how powerful this technique is: we get all the perks of native Rust tasks with the optimizations and ergonomics of global state. This means your *actual* state does not need to be tied up in a system like `Signal::global` or Redux – the only Atoms that need to exist are those that are used to drive the display/UI.

```
static USERNAME: GlobalSignal<String> = Signal::global(|| "default".to

fn app() -> Element {
    use_coroutine(sync_service);

    rsx! { Banner {} }
}

fn Banner() -> Element {
    rsx! { h1 { "Welcome back, {USERNAME}" } }
}
```

Now, in our sync service, we can structure our state however we want. We only need to update the view values when ready.

```
use futures_util::StreamExt;

static USERNAME: GlobalSignal<String> = Signal::global(|| "default".to_string
static ERRORS: GlobalSignal<Vec<String>> = Signal::global(|| Vec::new());

enum SyncAction {
    SetUsername(String),
}
```

```
async fn sync_service(mut rx: UnboundedReceiver<SyncAction>) {
    while let Some(msg) = rx.next().await {
        match msg {
            SyncAction::SetUsername(name) => {
                if set_name_on_server(&name).await.is_ok() {
                    *USERNAME.write() = name;
                } else {
                    *ERRORS.write() = vec!["Failed to set username".to_string
                }
            }
        }
    }
}
```

Automatic injection into the Context API

Coroutine handles are automatically injected through the context API. You can use the `use_coroutine_handle` hook with the message type as a generic to fetch a handle.

```
fn Child() -> Element {
    let sync_task = use_coroutine_handle::<SyncAction>();

    sync_task.send(SyncAction::SetUsername);

    todo!()
}
```

Copy

Resource

`use_resource` lets you run an async closure, and provides you with its result.

For example, we can make an API request (using reqwest) inside `use_resource`:

```
let mut future = use_resource(|| async move {
    reqwest::get("https://dog.ceo/api/breeds/image/random")
        .await
        .unwrap()
        .json::<ApiResponse>()
        .await
});
```

The code inside `use_resource` will be submitted to the Dioxus scheduler once the component has rendered.

We can use `.read()` to get the result of the future. On the first run, since there's no data ready when the component loads, its value will be `None`. However, once the future is finished, the component will be re-rendered and the value will now be `Some(...)`, containing the return value of the closure.

We can then render that result:

```
match &*future.read_unchecked() {
    Some(Ok(response)) => rsx! {
        button { onclick: move |_| future.restart(), "Click to fetch another
        div { img {
            max_width: "500px",
            max_height: "500px",
            src: "{response.image_url}"
        } }
    },
    Some(Err(_)) => rsx! { div { "Loading dogs failed" } },
    None => rsx! { div { "Loading dogs..." } },
}
```

Loading dogs...

Restarting the Future

The `Resource` handle provides a `restart` method. It can be used to execute the future again, producing a new value.

Dependencies

Often, you will need to run the future again every time some value (e.g. a state) changes. Rather than calling `restart` manually, you can read a signal inside of the future. It will automatically re-run the future when any of the states you read inside the future change. Example:

```
let future = use_resource(move || async move {
    reqwest::get(format!("https://dog.ceo/api/breed/{breed}/images/random"))
        .await
        .unwrap()
        .json::<ApiResponse>()
        .await
});

// You can also add non-reactive state to the resource hook with the use_reac
let non_reactive_state = "poodle";
use_resource(use_reactive!(|(non_reactive_state,)| async move {
    reqwest::get(format!(
        "https://dog.ceo/api/breed/{non_reactive_state}/images/random"
    ))
    .await
    .unwrap()
    .json::<ApiResponse>()
    .await
}));
```

Router

In many of your apps, you'll want to have different "scenes". For a webpage, these scenes might be the different webpages with their own content. For a desktop app, these scenes might be different views in your app.

To unify these platforms, Dioxus provides a first-party solution for scene management called Dioxus Router.

What is it?

For an app like the Dioxus landing page (https://dioxuslabs.com), we want to have several different scenes:

- Homepage
- Blog

Each of these scenes is independent – we don't want to render both the homepage and blog at the same time.

The Dioxus router makes it easy to create these scenes. To make sure we're using the router, add the `router` feature to your `dioxus` dependency:

```
cargo add dioxus@0.5.0 --features router
```
Copy

Using the router

Unlike other routers in the Rust ecosystem, our router is built declaratively at compile time. This makes it possible to compose our app layout simply by defining an enum.

```rust
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    // if the current location is "/home", render the Home component
    #[route("/home")]
    Home {},
    // if the current location is "/blog", render the Blog component
    #[route("/blog")]
    Blog {},
}

fn Home() -> Element {
    todo!()
}

fn Blog() -> Element {
    todo!()
}
```
Copy

Whenever we visit this app, we will get either the Home component or the Blog component rendered depending on which route we enter at. If neither of these routes match the current location, then nothing will render.

We can fix this one of two ways:

- A fallback 404 page

```rust
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[route("/home")]
    Home {},
    #[route("/blog")]
    Blog {},
    //  if the current location doesn't match any of the above routes, render
    #[route("/:..segments")]
    NotFound { segments: Vec<String> },
}

fn Home() -> Element {
    todo!()
}

fn Blog() -> Element {
    todo!()
}
```
Copy

```rust
#[component]
fn NotFound(segments: Vec<String>) -> Element {
    todo!()
}
```

- Redirect 404 to home

```rust
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[route("/home")]
    //  if the current location doesn't match any of the other routes, redire
    #[redirect("/:..segments", |segments: Vec<String>| Route::Home {})]
    Home {},
    #[route("/blog")]
    Blog {},
}
```

Links

For our app to navigate these routes, we can provide clickable elements called Links. These simply wrap `<a>` elements that, when clicked, navigate the app to the given location. Because our route is an enum of valid routes, if you try to link to a page that doesn't exist, you will get a compiler error.

```rust
rsx! {
    Link { to: Route::Home {}, "Go home!" }
}
```

More reading

This page is just a very brief overview of the router. For more information, check out the router book or some of the router examples.

Dynamic Rendering

Sometimes you want to render different things depending on the state/props. With Dioxus, just describe what you want to see using Rust control flow – the framework will take care of making the necessary changes on the fly if the state or props change!

Conditional Rendering

To render different elements based on a condition, you could use an `if-else` statement:

```
if is_logged_in {
    rsx! {
        "Welcome!"
        button { onclick: move |_| log_out.call(()), "Log Out" }
    }
} else {
    rsx! { button { onclick: move |_| log_in.call(()), "Log In" } }
}
```

Log In

> You could also use `match` statements, or any Rust function to conditionally render different things.

Improving the `if-else` Example

You may have noticed some repeated code in the `if-else` example above. Repeating code like this is both bad for maintainability and performance. Dioxus will skip diffing static elements like the button, but when switching between multiple `rsx` calls it cannot perform this optimization. For this example either approach is fine, but for components with large parts that are reused between conditionals, it can be more of an issue.

We can improve this example by splitting up the dynamic parts and inserting them where they are needed.

```
rsx! {
    // We only render the welcome message if we are logged in
    // You can use if statements in the middle of a render block to condition
    if is_logged_in {
        // Notice the body of this if statement is rsx code, not an expressio
        "Welcome!"
    }
    button {
        // depending on the value of `is_logged_in`, we will call a differen
        onclick: move |_| if is_logged_in { log_out.call(()) } else { log_in
        if is_logged_in {
            // if we are logged in, the button should say "Log Out"
            "Log Out"
        } else {
            // if we are not logged in, the button should say "Log In"
            "Log In"
        }
    }
}
```

Log In

Inspecting `Element` props

Since `Element` is a `Option<VNode>`, components accepting `Element` as a prop can inspect its contents, and render different things based on that. Example:

```
fn Clickable(props: ClickableProps) -> Element {
    match props.children {
        Some(VNode { .. }) => {
```

On this page

Edit this page!

Go to version
0.5
0.4
0.3

```
            todo!("render some stuff")
        }
        _ => {
            todo!("render some other stuff")
        }
    }
}
```

You can't mutate the `Element`, but if you need a modified version of it, you can construct a new one based on its attributes/children/etc.

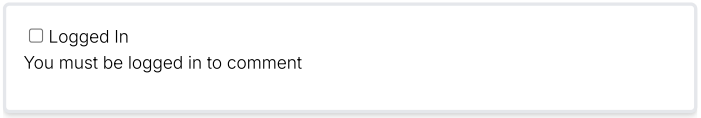
Rendering Nothing

To render nothing, you can return `None` from a component. This is useful if you want to conditionally hide something:

```
if is_logged_in {
    return None;
}

rsx! { p { "You must be logged in to comment" } }
```

Copy



This works because the `Element` type is just an alias for `Option<VNode>`

> Again, you may use a different method to conditionally return `None`. For example the boolean's `then()` function could be used.

Rendering Lists

Often, you'll want to render a collection of components. For example, you might want to render a list of all comments on a post.

For this, Dioxus accepts iterators that produce `Element`s. So we need to:

- Get an iterator over all of our items (e.g., if you have a `Vec` of comments, iterate over it with `iter()`)
- `.map` the iterator to convert each item into a `LazyNode` using `rsx!{...}`
  - Add a unique `key` attribute to each iterator item
- Include this iterator in the final RSX (or use it inline)

Example: suppose you have a list of comments you want to render. Then, you can render them like this:

```
let mut comment_field = use_signal(String::new);
let mut next_id = use_signal(|| 0);
let mut comments = use_signal(Vec::<Comment>::new);

let comments_lock = comments.read();
let comments_rendered = comments_lock.iter().map(|comment| {
    rsx! { CommentComponent { comment: comment.clone() } }
});

rsx! {
    form {
        onsubmit: move |_| {
            comments
                .write()
                .push(Comment {
                    content: comment_field(),
                    id: next_id(),
                });
            next_id += 1;
            comment_field.set(String::new());
        },
        input {
            value: "{comment_field}",
            oninput: move |event| comment_field.set(event.value())
        }
        input { r#type: "submit" }
    }
    {comments_rendered}
}
```

Copy

Inline for loops

Because of how common it is to render a list of items, Dioxus provides a shorthand for this. Instead of using `.iter`, `.map`, and `rsx`, you can use a `for` loop with a body of rsx code:

```
let mut comment_field = use_signal(String::new);
let mut next_id = use_signal(|| 0);
let mut comments = use_signal(Vec::<Comment>::new);

rsx! {
    form {
        onsubmit: move |_| {
            comments
                .write()
                .push(Comment {
                    content: comment_field(),
                    id: next_id(),
                });
            next_id += 1;
            comment_field.set(String::new());
        },
        input {
            value: "{comment_field}",
            oninput: move |event| comment_field.set(event.value())
        }
        input { r#type: "submit" }
    }
    for comment in comments() {
        // Notice the body of this for loop is rsx code, not an expression
        CommentComponent { comment }
    }
}
```

Copy

Submit

The `key` Attribute

Every time you re-render your list, Dioxus needs to keep track of which items go where to determine what updates need to be made to the UI.

For example, suppose the `CommentComponent` had some state – e.g. a field where the user typed in a reply. If the order of comments suddenly changes, Dioxus needs to correctly associate that state with the same comment – otherwise, the user will end up replying to a different comment!

To help Dioxus keep track of list items, we need to associate each item with a unique key. In the example above, we dynamically generated the unique key. In real applications, it's more likely that the key will come from e.g. a database ID. It doesn't matter where you get the key from, as long as it meets the requirements:

- Keys must be unique in a list
- The same item should always get associated with the same key
- Keys should be relatively small (i.e. converting the entire Comment structure to a String would be a pretty bad key) so they can be compared efficiently

You might be tempted to use an item's index in the list as its key. That's what Dioxus will use if you don't specify a key at all. This is only acceptable if you can guarantee that the list is constant – i.e., no re-ordering, additions, or deletions.

> Note that if you pass the key to a component you've made, it won't receive the key as a prop. It's only used as a hint by Dioxus itself. If your component needs an ID, you have to pass it as a separate prop.

Dioxus Labs

An Open Source project dedicated to making Rust UI wonderful.

Community

Github
Twitter
Discord

Learning

docs.rs
Guide
Awesome

Projects

Dioxus
CLI
Taffy

Sharing State

Often, multiple components need to access the same state. Depending on your needs, there are several ways to implement this.

Lifting State

One approach to share state between components is to "lift" it up to the nearest common ancestor. This means putting the `use_signal` hook in a parent component, and passing the needed values down as props.

Suppose we want to build a meme editor. We want to have an input to edit the meme caption, but also a preview of the meme with the caption. Logically, the meme and the input are 2 separate components, but they need access to the same state (the current caption).

> Of course, in this simple example, we could write everything in one component – but it is better to split everything out in smaller components to make the code more reusable, maintainable, and performant (this is even more important for larger, complex apps).

We start with a `Meme` component, responsible for rendering a meme with a given caption:

```
#[component]
fn Meme(caption: String) -> Element {
    let container_style = r#"
        position: relative;
        width: fit-content;
    "#;

    let caption_container_style = r#"
        position: absolute;
        bottom: 0;
        left: 0;
        right: 0;
        padding: 16px 8px;
    "#;

    let caption_style = r"
        font-size: 32px;
        margin: 0;
        color: white;
        text-align: center;
    ";

    rsx! {
        div { style: "{container_style}",
            img { src: "https://i.imgflip.com/2zh47r.jpg", height: "500px" }
            div { style: "{caption_container_style}", p { style: "{caption_st
        }
    }
}
```

> Note that the `Meme` component is unaware where the caption is coming from – it could be stored in `use_signal`, or a constant. This ensures that it is very reusable – the same component can be used for a meme gallery without any changes!

We also create a caption editor, completely decoupled from the meme. The caption editor must not store the caption itself – otherwise, how will we provide it to the `Meme` component? Instead, it should accept the current caption as a prop, as well as an event handler to delegate input events to:

```
#[component]
fn CaptionEditor(caption: String, oninput: EventHandler<FormEvent>) -> Elemer
    let input_style = r"
        border: none;
        background: cornflowerblue;
        padding: 8px 16px;
        margin: 0;
        border-radius: 4px;
        color: white;
    ";

    rsx! {
        input {
            style: "{input_style}",
```

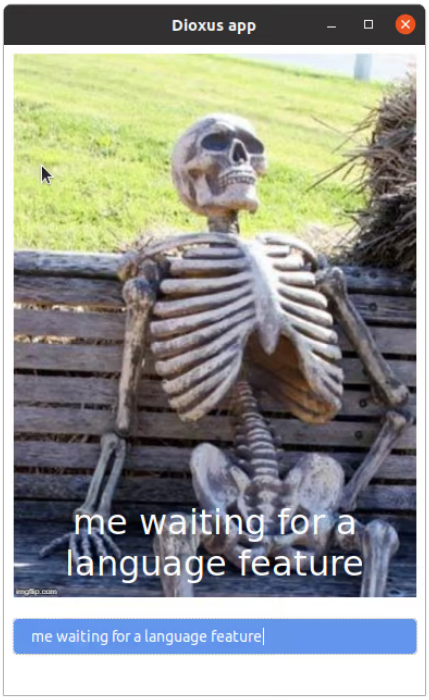
```
            value: "{caption}",
            oninput: move |event| oninput.call(event)
        }
    }
}
```

Finally, a third component will render the other two as children. It will be responsible for keeping the state and passing down the relevant props.

```
fn MemeEditor() -> Element {
    let container_style = r"
        display: flex;
        flex-direction: column;
        gap: 16px;
        margin: 0 auto;
        width: fit-content;
    ";

    let mut caption = use_signal(|| "me waiting for my rust code to compile",

    rsx! {
        div { style: "{container_style}",
            h1 { "Meme Editor" }
            Meme { caption: caption }
            CaptionEditor { caption: caption, oninput: move |event: FormEvent
        }
    }
}
```



Using Shared State

Sometimes, some state needs to be shared between multiple components far down the tree, and passing it down through props is very inconvenient.

Suppose now that we want to implement a dark mode toggle for our app. To achieve this, we will make every component select styling depending on whether dark mode is enabled or not.

> Note: we're choosing this approach for the sake of an example. There are better ways to implement dark mode (e.g. using CSS variables). Let's pretend CSS variables don't exist – welcome to 2013!

Now, we could write another `use_signal` in the top component, and pass `is_dark_mode` down to every component through props. But think about what will happen as the app grows in complexity – almost every component that renders any CSS is going to need to know if dark mode is enabled or not – so they'll all need the same dark mode prop. And every parent component will need to pass it down to them. Imagine how messy and verbose that would get, especially if we had components several levels deep!

Dioxus offers a better solution than this "prop drilling" – providing context. The `use_context_provider` hook provides any Clone context (including Signals!) to any

child components. Child components can use the `use_context` hook to get that context and if it is a Signal, they can read and write to it.

First, we have to create a struct for our dark mode configuration:

```
#[derive(Clone, Copy)]
struct DarkMode(bool);
```

Copy

Now, in a top-level component (like `App`), we can provide the `DarkMode` context to all children components:

```
use_context_provider(|| Signal::new(DarkMode(false)));
```

Copy

As a result, any child component of `App` (direct or not), can access the `DarkMode` context.

```
let dark_mode_context = use_context::<Signal<DarkMode>>();
```

Copy

> `use_context` returns `Signal<DarkMode>` here, because the Signal was provided by the parent. If the context hadn't been provided `use_context` would have panicked.

If you have a component where the context might or not be provided, you might want to use `try_consume_context` instead, so you can handle the `None` case. The drawback of this method is that it will not memoize the value between renders, so it won't be as as efficient as `use_context`, you could do it yourself with `use_hook` though.

For example, here's how we would implement the dark mode toggle, which both reads the context (to determine what color it should render) and writes to it (to toggle dark mode):

```
pub fn DarkModeToggle() -> Element {
    let mut dark_mode = use_context::<Signal<DarkMode>>();

    let style = if dark_mode().0 { "color:white" } else { "" };

    rsx! {
        label { style: "{style}",
            "Dark Mode"
            input {
                r#type: "checkbox",
                oninput: move |event| {
                    let is_enabled = event.value() == "true";
                    dark_mode.write().0 = is_enabled;
                }
            }
        }
    }
}
```

Copy

User Input

Interfaces often need to provide a way to input data: e.g. text, numbers, checkboxes, etc. In Dioxus, there are two ways you can work with user input.

Controlled Inputs

With controlled inputs, you are directly in charge of the state of the input. This gives you a lot of flexibility, and makes it easy to keep things in sync. For example, this is how you would create a controlled text input:

```
pub fn App() -> Element {
    let mut name = use_signal(|| "bob".to_string());

    rsx! {
        input {
            // we tell the component what to render
            value: "{name}",
            // and what to do when the value changes
            oninput: move |event| name.set(event.value())
        }
    }
}
```

Copy

bob

Notice the flexibility – you can:

- Also display the same contents in another element, and they will be in sync
- Transform the input every time it is modified (e.g. to make sure it is upper case)
- Validate the input every time it changes
- Have custom logic happening when the input changes (e.g. network request for autocompletion)
- Programmatically change the value (e.g. a "randomize" button that fills the input with nonsense)

Uncontrolled Inputs

As an alternative to controlled inputs, you can simply let the platform keep track of the input values. If we don't tell a HTML input what content it should have, it will be editable anyway (this is built into the browser). This approach can be more performant, but less flexible. For example, it's harder to keep the input in sync with another element.

Since you don't necessarily have the current value of the uncontrolled input in state, you can access it either by listening to `oninput` events (similarly to controlled components), or, if the input is part of a form, you can access the form data in the form events (e.g. `oninput` or `onsubmit`):

```
pub fn App() -> Element {
    rsx! {
        form { onsubmit: move |event| { log::info!("Submitted! {event:?}") },
            input { name: "name" }
            input { name: "age" }
            input { name: "date" }
            input { r#type: "submit" }
        }
    }
}
```

Copy

Submit

```
Submitted! UiEvent { data: FormData { value: "", values: {"age": "very
```

Copy

Handling files

You can insert a file picker by using an input element of type `file`. This element supports the `multiple` attribute, to let you pick more files at the same time. You can select a folder by adding the `directory` attribute: Dioxus will map this attribute to browser specific attributes, because there is no standardized way to allow a directory to be selected.

`type` is a Rust keyword, so when specifying the type of the input field, you have to write it as `r#type:"file"`.

Extracting the selected files is a bit different from what you may typically use in Javascript.

The `FormData` event contains a `files` field with data about the uploaded files. This field contains a `FileEngine` struct which lets you fetch the filenames selected by the user. This example saves the filenames of the selected files to a `Vec`:

```
pub fn App() -> Element {
    let mut filenames: Signal<Vec<String>> = use_signal(Vec::new);
    rsx! {
        input {
            // tell the input to pick a file
            r#type: "file",
            // list the accepted extensions
            accept: ".txt,.rs",
            // pick multiple files
            multiple: true,
            onchange: move |evt| {
                if let Some(file_engine) = &evt.files() {
                    let files = file_engine.files();
                    for file_name in files {
                        filenames.write().push(file_name);
                    }
                }
            }
        }
    }
}
```

If you're planning to read the file content, you need to do it asynchronously, to keep the rest of the UI interactive. This example event handler loads the content of the selected files in an async closure:

```
onchange: move |evt| {
    async move {
        if let Some(file_engine) = evt.files() {
            let files = file_engine.files();
            for file_name in &files {
                if let Some(file) = file_engine.read_file_to_string(file_name
                {
                    files_uploaded.write().push(file);
                }
            }
        }
    }
}
```

Lastly, this example shows you how to select a folder, by setting the `directory` attribute to `true`.

```
input {
    r#type: "file",
    // Select a folder by setting the directory attribute
    directory: true,
    onchange: move |evt| {
        if let Some(file_engine) = evt.files() {
            let files = file_engine.files();
            for file_name in files {
                println!("{}", file_name);
            }
        }
    }
}
```

Dioxus Labs

An Open Source project dedicated to making Rust UI wonderful.

Community

Github
Twitter
Discord

Learning

docs.rs
Guide
Awesome

Projects

Dioxus
CLI
Taffy

Hooks and component state

So far, our components have had no state like a normal Rust function. However, in a UI component, it is often useful to have stateful functionality to build user interactions. For example, you might want to track whether the user has opened a drop-down and render different things accordingly.

Hooks allow us to create state in our components. Hooks are Rust functions you call in a constant order in a component that add additional functionality to the component.

Dioxus provides many built-in hooks, but if those hooks don't fit your specific use case, you also can create your own hook

use_signal hook

`use_signal` is one of the simplest hooks.

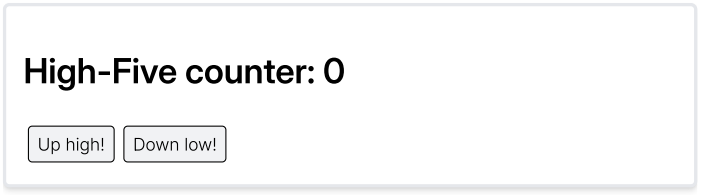
- You provide a closure that determines the initial value: `let mut count = use_signal(|| 0);`
- `use_signal` gives you the current value, and a way to write to the value
- When the value updates, `use_signal` makes the component re-render (along with any other component that references it), and then provides you with the new value.

For example, you might have seen the counter example, in which state (a number) is tracked using the `use_signal` hook:

```
pub fn App() -> Element {
    // count will be initialized to 0 the first time the component is rendere
    let mut count = use_signal(|| 0);

    rsx! {
        h1 { "High-Five counter: {count}" }
        button { onclick: move |_| count += 1, "Up high!" }
        button { onclick: move |_| count -= 1, "Down low!" }
    }
}
```

Copy



Every time the component's state changes, it re-renders, and the component function is called, so you can describe what you want the new UI to look like. You don't have to worry about "changing" anything – describe what you want in terms of the state, and Dioxus will take care of the rest!

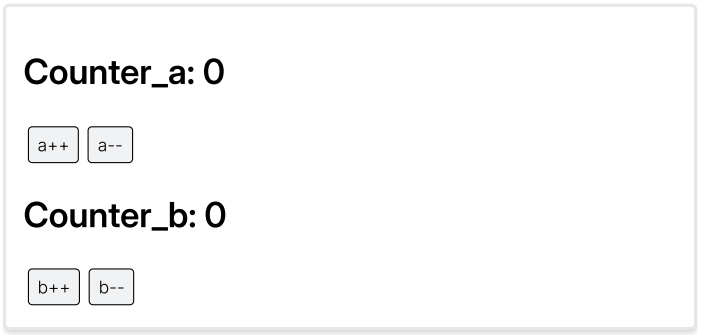
> `use_signal` returns your value wrapped in a smart pointer of type `Signal` that is `Copy`. This is why you can both read the value and update it, even within an event handler.

You can use multiple hooks in the same component if you want:

```
pub fn App() -> Element {
    let mut count_a = use_signal(|| 0);
    let mut count_b = use_signal(|| 0);

    rsx! {
        h1 { "Counter_a: {count_a}" }
        button { onclick: move |_| count_a += 1, "a++" }
        button { onclick: move |_| count_a -= 1, "a--" }
        h1 { "Counter_b: {count_b}" }
        button { onclick: move |_| count_b += 1, "b++" }
        button { onclick: move |_| count_b -= 1, "b--" }
    }
}
```

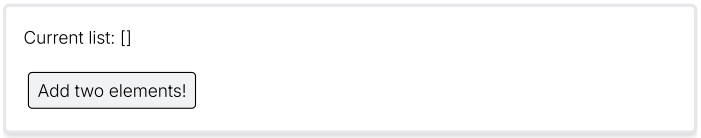
Copy



You can also use `use_signal` to store more complex state, like a Vec. You can read and write to the state with the `read` and `write` methods:

```
pub fn App() -> Element {
    let mut list = use_signal(Vec::new);

    rsx! {
        p { "Current list: {list:?}" }
        button {
            onclick: move |event| {
                let list_len = list.len();
                list.push(list_len);
                list.push(list_len);
            },
            "Add two elements!"
        }
    }
}
```



Rules of hooks

The above example might seem a bit magic since Rust functions are typically not associated with state. Dioxus allows hooks to maintain state across renders through a hidden scope that is associated with the component.

But how can Dioxus differentiate between multiple hooks in the same component? As you saw in the second example, both `use_signal` functions were called with the same parameters, so how come they can return different things when the counters are different?

```
let mut count_a = use_signal(|| 0);
let mut count_b = use_signal(|| 0);
```

This is only possible because the two hooks are always called in the same order, so Dioxus knows which is which. Because the order you call hooks matters, you must follow certain rules when using hooks:

1. Hooks may be only used in components or other hooks (we'll get to that later).
2. On every call to a component function.
3. The same hooks must be called (except in the case of early returns, as explained later in the Error Handling chapter).
4. In the same order.
5. Hook names should start with `use_` so you don't accidentally confuse them with regular functions (`use_signal()`, `use_signal()`, `use_resource()`, etc...).

These rules mean that there are certain things you can't do with hooks:

No hooks in conditionals

```
// ❌ don't call hooks in conditionals!
// We must ensure that the same hooks will be called every time
// But `if` statements only run if the conditional is true!
// So we might violate rule 2.
if you_are_happy && you_know_it {
    let something = use_signal(|| "hands");
    println!("clap your {something}")
}

// ✅ instead, *always* call use_signal
```

```rust
// You can put other stuff in the conditional though
let something = use_signal(|| "hands");
if you_are_happy && you_know_it {
    println!("clap your {something}")
}
```

No hooks in closures

Copy

```rust
// ❌ don't call hooks inside closures!
// We can't guarantee that the closure, if used, will be called in the same o
let _a = || {
    let b = use_signal(|| 0);
    b()
};

// ✅ instead, move hook `b` outside
let b = use_signal(|| 0);
let _a = || b();
```

No hooks in loops

Copy

```rust
// `names` is a Vec<&str>

// ❌ Do not use hooks in loops!
// In this case, if the length of the Vec changes, we break rule 2
for _name in &names {
    let is_selected = use_signal(|| false);
    println!("selected: {is_selected}");
}

// ✅ Instead, use a hashmap with use_signal
let selection_map = use_signal(HashMap::<&str, bool>::new);

for name in &names {
    let is_selected = selection_map.read()[name];
    println!("selected: {is_selected}");
}
```

Additional resources

- dioxus_hooks API docs
- dioxus_hooks source code