

Calling Rust from the frontend

Tauri provides a simple yet powerful `command` system for calling Rust functions from your web app. Commands can accept arguments and return values. They can also return errors and be `async`.

Basic Example

Commands are defined in your `src-tauri/src/main.rs` file. To create a command, just add a function and annotate it with `#[tauri::command]`:

```
#[tauri::command]
fn my_custom_command() {
    println!("I was invoked from JS!");
}
```

You will have to provide a list of your commands to the builder function like so:

```
// Also in main.rs
fn main() {
    tauri::Builder::default()
        // This is where you pass in your commands
        .invoke_handler(tauri::generate_handler![my_custom_command])
        .run(tauri::generate_context!())
        .expect("failed to run app");
}
```

Now, you can invoke the command from your JS code:

```
// When using the Tauri API npm package:
import { invoke } from '@tauri-apps/api/tauri'

// When using the Tauri global script (if not using the npm package)
// Be sure to set `build.withGlobalTauri` in `tauri.conf.json` to true
const invoke = window.__TAURI__.invoke

// Invoke the command
invoke('my_custom_command')
```

Passing Arguments

Your command handlers can take arguments:

```
#[tauri::command]
fn my_custom_command(invoker: String) {
    println!("I was invoked from JS, with this message: {}", invoker);
}
```

Arguments should be passed as a JSON object with camelCase keys:

```
invoke('my_custom_command', { invokeMessage: 'Hello!' })
```

Arguments can be of any type, as long as they implement `serde::Deserialize`.

Please note, when declaring arguments in Rust using `snake_case`, the arguments are converted to `camelCase` for JavaScript.

To use `snake_case` in JavaScript, you have to declare it in the `tauri::command` statement:

```
#[tauri::command(rename_all = "snake_case")]
fn my_custom_command(invoker: String) {
    println!("I was invoked from JS, with this message: {}", invoker);
}
```

The corresponding JavaScript:

```
invoke('my_custom_command', { invoke_message: 'Hello!' })
```

Returning Data

Command handlers can return data as well:

```
#[tauri::command]
fn my_custom_command() -> String {
    "Hello from Rust!".into()
}
```

The `invoke` function returns a promise that resolves with the returned value:

```
invoke('my_custom_command').then((message) => console.log(message))
```

Returned data can be of any type, as long as it implements `serde::Serialize`.

Error Handling

If your handler could fail and needs to be able to return an error, have the function return a `Result`:

```
#[tauri::command]
fn my_custom_command() -> Result<String, String> {
    // If something fails
    Err("This failed!".into())
    // If it worked
    Ok("This worked!".into())
}
```

If the command returns an error, the promise will reject, otherwise, it resolves:

```
invoke('my_custom_command')
    .then((message) => console.log(message))
    .catch((error) => console.error(error))
```

As mentioned above, everything returned from commands must implement `serde::Serialize`, including errors. This can be problematic if you're working with error types from Rust's std library or external crates as most error types do not implement it. In simple scenarios you can use `map_err` to convert these errors to `Strings`:

```
#[tauri::command]
fn my_custom_command() -> Result<(), String> {
    // This will return an error
    std::fs::File::open("path/that/does/not/exist").map_err(|err| err.to_string());
    // Return nothing on success
    Ok(())
}
```

Since this is not very idiomatic you may want to create your own error type which implements `serde::Serialize`. In the following example, we use the `thiserror` crate to help create the error type. It allows you to turn enums into error types by deriving the `thiserror::Error` trait. You can consult its documentation for more details.

```
// create the error type that represents all errors possible in our program
#[derive(Debug, thiserror::Error)]
enum Error {
    #[error(transparent)]
    Io(#[from] std::io::Error)
}

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

#[tauri::command]
fn my_custom_command() -> Result<(), Error> {
    // This will return an error
    std::fs::File::open("path/that/does/not/exist")?;
    // Return nothing on success
    Ok(())
}
```

A custom error type has the advantage of making all possible errors explicit so readers can quickly identify what errors can happen. This saves other people (and yourself) enormous amounts of time when reviewing and refactoring code later.

It also gives you full control over the way your error type gets serialized. In the above example, we simply returned the error message as a string, but you could assign each error a code similar to C, this way you could more easily map it to a similar looking TypeScript error enum for example.

Async Commands

Asynchronous functions are beneficial in Tauri to perform heavy work in a manner that doesn't result in UI freezes or slowdowns.

NOTE

Async commands are executed on a separate thread using `async_runtime::spawn`. Commands without the `async` keyword are executed on the main thread unless defined with `#[tauri::command(async)]`.

If your command needs to run asynchronously, simply declare it as `async`.

CAUTION

You need to be careful when creating asynchronous functions using Tauri. Currently, you cannot simply include borrowed arguments in the signature of an asynchronous function. Some common examples of types like this are `&str` and `State<'_, Data>`. This limitation is tracked here: <https://github.com/tauri-apps/tauri/issues/2533> and workarounds are shown below.

When working with borrowed types, you have to make additional changes. These are your two main options:

Option 1: Convert the type, such as `&str` to a similar type that is not borrowed, such as `String`. This may not work for all types, for example `State<'_, Data>`.

Example:

```
// Declare the async function using String instead of &str, as &str is borrowed and thus unsupported
#[tauri::command]
async fn my_custom_command(value: String) -> String {
    // Call another async function and wait for it to finish
    some_async_function().await;
    format!(value)
}
```

Option 2: Wrap the return type in a `Result`. This one is a bit harder to implement, but should work for all types.

Use the return type `Result<a, b>`, replacing `a` with the type you wish to return, or `()` if you wish to return nothing, and replacing `b` with an error type to return if something goes wrong, or `()` if you wish to have no optional error returned. For example:

- `Result<String, ()>` to return a `String`, and no error.
- `Result<(), ()>` to return nothing.
- `Result<bool, Error>` to return a boolean or an error as shown in the [Error Handling](#) section above.

Example:

```
// Return a Result<String, ()> to bypass the borrowing issue
#[tauri::command]
async fn my_custom_command(value: &str) -> Result<String, ()> {
```

```
// Call another async function and wait for it to finish
some_async_function().await;
// Note that the return value must be wrapped in `Ok()` now.
Ok(format!(value))
}
```

Invoking from JS

Since invoking the command from JavaScript already returns a promise, it works just like any other command:

```
invoke('my_custom_command', { value: 'Hello, Async!' }).then(() =>
  console.log('Completed!')
)
```

Accessing the Window in Commands

Commands can access the `Window` instance that invoked the message:

```
#[tauri::command]
async fn my_custom_command(window: tauri::Window) {
  println!("Window: {}", window.label());
}
```

Accessing an AppHandle in Commands

Commands can access an `AppHandle` instance:

```
#[tauri::command]
async fn my_custom_command(app_handle: tauri::AppHandle) {
  let app_dir = app_handle.path_resolver().app_dir();
  use tauri::GlobalShortcutManager;
  app_handle.global_shortcut_manager().register("CTRL + U", move || {});
}
```

Accessing managed state

Tauri can manage state using the `manage` function on `tauri::Builder`. The state can be accessed on a command using `tauri::State`:

```
struct MyState(String);

#[tauri::command]
fn my_custom_command(state: tauri::State<MyState>) {
    assert_eq!(state.0 == "some state value", true);
}

fn main() {
    tauri::Builder::default()
        .manage(MyState("some state value".into()))
        .invoke_handler(tauri::generate_handler![my_custom_command])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Creating Multiple Commands

The `tauri::generate_handler!` macro takes an array of commands. To register multiple commands, you cannot call `invoke_handler` multiple times. Only the last call will be used. You must pass each command to a single call of `tauri::generate_handler!`.

```
#[tauri::command]
fn cmd_a() -> String {
    "Command a"
}

#[tauri::command]
fn cmd_b() -> String {
    "Command b"
}

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

Complete Example

Any or all of the above features can be combined:

```
struct Database;

#[derive(serde::Serialize)]
struct CustomResponse {
    message: String,
    other_val: usize,
}

async fn some_other_function() -> Option<String> {
    Some("response".into())
}

#[tauri::command]
async fn my_custom_command(
    window: tauri::Window,
    number: usize,
    database: tauri::State<'_, Database>,
) -> Result<CustomResponse, String> {
    println!("Called from {}", window.label());
    let result: Option<String> = some_other_function().await;
    if let Some(message) = result {
        Ok(CustomResponse {
            message,
            other_val: 42 + number,
        })
    } else {
        Err("No result".into())
    }
}


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

```
// Invocation from JS
```

```
invoke('my_custom_command', {
  number: 42,
})
.then((res) =>
  console.log(`Message: ${res.message}, Other Val: ${res.other_val}`)
```



```
)  
.catch((e) => console.error(e))
```

 [Edit this page](#)

Last updated on May 22, 2023