# Application Debugging

With all the moving pieces in Tauri, you may run into a problem that requires debugging. There are many locations where error details are printed, and Tauri includes some tools to make the debugging process more straightforward.

## Rust Console

The first place to look for errors is in the Rust Console. This is in the terminal where you ran, e.g., `tauri dev`. You can use the following code to print something to that console from within a Rust file:

```
println!("Message from Rust: {}", msg);
```

Sometimes you may have an error in your Rust code, and the Rust compiler can give you lots of information. If, for example, `tauri dev` crashes, you can rerun it like this on Linux and macOS:

```
$ RUST_BACKTRACE=1 tauri dev
```

or like this on Windows:

```
$ set RUST_BACKTRACE=1
$ tauri dev
```

This command gives you a granular stack trace. Generally speaking, the Rust compiler helps you by giving you detailed information about the issue, such as:

```
error[E0425]: cannot find value `sun` in this scope
  --> src/main.rs:11:5
   |
11 |     sun += i.to_string().parse::<u64>().unwrap();
   |     ^^^ help: a local variable with a similar name exists: `sum`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0425`.
```

# WebView Console

Right-click in the WebView, and choose `Inspect Element`. This opens up a web-inspector similar to the Chrome or Firefox dev tools you are used to. You can also use the `Ctrl + Shift + i` shortcut on Linux and Windows, and `Command + Option + i` on macOS to open the inspector.

The inspector is platform-specific, rendering the webkit2gtk WebInspector on Linux, Safari's inspector on macOS and the Microsoft Edge DevTools on Windows.

## Opening Devtools Programmatically

You can control the inspector window visibility by using the `Window::open_devtools` and `Window::close_devtools` functions:

```
use tauri::Manager;
tauri::Builder::default()
  .setup(|app| {
    #[cfg(debug_assertions)] // only include this code on debug builds
    {
      let window = app.get_window("main").unwrap();
      window.open_devtools();
      window.close_devtools();
    }
    Ok(())
  });
```

## Using the Inspector in Production

By default, the inspector is only enabled in development and debug builds unless you enable it with a Cargo feature.

### Create a Debug Build

To create a debug build, run the `tauri build --debug` command.

npm Yarn pnpm Cargo

```
$ cargo tauri build --debug
```

Like the normal build and dev processes, building takes some time the first time you run this command but is significantly faster on subsequent runs. The final bundled app has the development console enabled and is placed in `src-tauri/target/debug/bundle`.

You can also run a built app from the terminal, giving you the Rust compiler notes (in case of errors) or your `println` messages. Browse to the file `src-tauri/target/(release|debug)/[app name]` and run it in directly in your console or double-click the executable itself in the filesystem (note: the console closes on errors with this method).

**Enable Devtools Feature**

> **DANGER**
>
> The devtools API is private on macOS. Using private APIs on macOS prevents your application from being accepted to the App Store.

To enable the devtools in production builds, you must enable the `devtools` Cargo feature in the `src-tauri/Cargo.toml` file:

```
[dependencies]
tauri = { version = "...", features = ["...", "devtools"] }
```

# Debugging the Core Process

The Core process is powered by Rust so you can use GDB or LLDB to debug it. You can follow the Debugging in VS Code guide to learn how to use the LLDB VS Code Extension to debug the Core Process of Tauri applications.