

Rust and WebAssembly

This small book describes how to use [Rust](#) and [WebAssembly](#) together.

Who is this book for?

This book is for anyone interested in compiling Rust to WebAssembly for fast, reliable code on the Web. You should know some Rust, and be familiar with JavaScript, HTML, and CSS. You don't need to be an expert in any of them.

Don't know Rust yet? Start with *The Rust Programming Language* first.


Don't know JavaScript, HTML, or CSS? [Learn about them on MDN](#).

How to read this book

You should read [the motivation for using Rust and WebAssembly together](#), as well as familiarize yourself with the [background and concepts](#) first.

The [tutorial](#) is written to be read from start to finish. You should follow along: writing, compiling, and running the tutorial's code yourself. If you haven't used Rust and WebAssembly together before, do the tutorial!

The [reference sections](#) may be perused in any order.

 **Tip:** You can search through this book by clicking on the  icon at the top of the page, or by pressing the `s` key.

Contributing to this book

This book is open source! Find a typo? Did we overlook something? [Send us a pull request!](#)

Why Rust and WebAssembly?

Low-Level Control with High-Level Ergonomics

JavaScript Web applications struggle to attain and retain reliable performance. JavaScript's dynamic type system and garbage collection pauses don't help. Seemingly small code changes can result in drastic performance regressions if you accidentally wander off the JIT's happy path.

Rust gives programmers low-level control and reliable performance. It is free from the non-deterministic garbage collection pauses that plague JavaScript. Programmers have control over indirection, monomorphization, and memory layout.

Small `.wasm` Sizes

Code size is incredibly important since the `.wasm` must be downloaded over the network. Rust lacks a runtime, enabling small `.wasm` sizes because there is no extra bloat included like a garbage collector. You only pay (in code size) for the functions you actually use.

Do *Not* Rewrite Everything

Existing code bases don't need to be thrown away. You can start by porting your most performance-sensitive JavaScript functions to Rust to gain immediate benefits. And you can even stop there if you want to.

Plays Well With Others

Rust and WebAssembly integrates with existing JavaScript tooling. It supports ECMAScript modules and you can continue using the tooling you already love, like npm and Webpack.

The Amenities You Expect

Rust has the modern amenities that developers have come to expect, such as:

- strong package management with `cargo`,
- expressive (and zero-cost) abstractions,

- and a welcoming community! 😊

Background and Concepts

This section provides the context necessary for diving into Rust and WebAssembly development.

What is WebAssembly?

WebAssembly (wasm) is a simple machine model and executable format with an [extensive specification](#). It is designed to be portable, compact, and execute at or near native speeds.

As a programming language, WebAssembly is comprised of two formats that represent the same structures, albeit in different ways:

1. The `.wat` text format (called `wat` for "**W**eb**A**ssembly **T**ext") uses [S-expressions](#), and bears some resemblance to the Lisp family of languages like Scheme and Clojure.
2. The `.wasm` binary format is lower-level and intended for consumption directly by wasm virtual machines. It is conceptually similar to ELF and Mach-O.

For reference, here is a factorial function in `wat`:

```
(module
  (func $fac (param f64) (result f64)
    local.get 0
    f64.const 1
    f64.lt
    if (result f64)
      f64.const 1
    else
      local.get 0
      local.get 0
      f64.const 1
      f64.sub
      call $fac
      f64.mul
    end)
  (export "fac" (func $fac)))
```

If you're curious about what a `wasm` file looks like you can use the [wat2wasm demo](#) with the above code.

Linear Memory

WebAssembly has a very simple [memory model](#). A wasm module has access to a single "linear memory", which is essentially a flat array of bytes. This [memory can be grown](#) by a multiple of the page size (64K). It cannot be shrunk.

Is WebAssembly Just for the Web?

Although it has currently gathered attention in the JavaScript and Web communities in general, wasm makes no assumptions about its host environment. Thus, it makes sense to speculate that wasm will become a "portable executable" format that is used in a variety of contexts in the future. As of *today*, however, wasm is mostly related to JavaScript (JS), which comes in many flavors (including both on the Web and [Node.js](#)).

Tutorial: Conway's Game of Life

This is a tutorial that implements [Conway's Game of Life](#) in Rust and WebAssembly.

Who is this tutorial for?

This tutorial is for anyone who already has basic Rust and JavaScript experience, and wants to learn how to use Rust, WebAssembly, and JavaScript together.

You should be comfortable reading and writing basic Rust, JavaScript, and HTML. You definitely do not need to be an expert.

What will I learn?

- How to set up a Rust toolchain for compiling to WebAssembly.
- A workflow for developing polyglot programs made from Rust, WebAssembly, JavaScript, HTML, and CSS.
- How to design APIs to take maximum advantage of both Rust and WebAssembly's strengths and also JavaScript's strengths.
- How to debug WebAssembly modules compiled from Rust.

- How to time profile Rust and WebAssembly programs to make them faster.
- How to size profile Rust and WebAssembly programs to make `.wasm` binaries smaller and faster to download over the network.

Setup

This section describes how to set up the toolchain for compiling Rust programs to WebAssembly and integrate them into JavaScript.

The Rust Toolchain

You will need the standard Rust toolchain, including `rustup`, `rustc`, and `cargo`.

[Follow these instructions to install the Rust toolchain.](#)

The Rust and WebAssembly experience is riding the Rust release trains to stable! That means we don't require any experimental feature flags. However, we do require Rust 1.30 or newer.

`wasm-pack`

`wasm-pack` is your one-stop shop for building, testing, and publishing Rust-generated WebAssembly.

Get `wasm-pack` [here!](#)

`cargo-generate`

`cargo-generate` helps you get up and running quickly with a new Rust project by leveraging a pre-existing git repository as a template.

Install `cargo-generate` with this command:

```
cargo install cargo-generate
```

`npm`

`npm` is a package manager for JavaScript. We will use it to install and run a JavaScript bundler and development server. At the end of the tutorial, we will publish our compiled `.wasm` to the `npm` registry.

Follow these instructions to install `npm`.

If you already have `npm` installed, make sure it is up to date with this command:

```
npm install npm@latest -g
```

Hello, World!

This section will show you how to build and run your first Rust and WebAssembly program: a Web page that alerts "Hello, World!"

Make sure you have followed the [setup instructions](#) before beginning.

Clone the Project Template

The project template comes pre-configured with sane defaults, so you can quickly build, integrate, and package your code for the Web.

Clone the project template with this command:

```
cargo generate --git https://github.com/rustwasm/wasm-pack-template
```

This should prompt you for the new project's name. We will use **"wasm-game-of-life"**.

```
wasm-game-of-life
```

What's Inside

Enter the new `wasm-game-of-life` project

```
cd wasm-game-of-life
```

and let's take a look at its contents:

```
wasm-game-of-life/  
├── Cargo.toml  
├── LICENSE_APACHE  
├── LICENSE_MIT  
├── README.md  
└── src  
    ├── lib.rs  
    └── utils.rs
```

Let's take a look at a couple of these files in detail.

wasm-game-of-life/Cargo.toml

The `Cargo.toml` file specifies dependencies and metadata for `cargo`, Rust's package manager and build tool. This one comes pre-configured with a `wasm-bindgen` dependency, a few optional dependencies we will dig into later, and the `crate-type` properly initialized for generating `.wasm` libraries.

wasm-game-of-life/src/lib.rs

The `src/lib.rs` file is the root of the Rust crate that we are compiling to WebAssembly. It uses `wasm-bindgen` to interface with JavaScript. It imports the `window.alert` JavaScript function, and exports the `greet` Rust function, which alerts a greeting message.

```
mod utils;  
  
use wasm_bindgen::prelude::*;  
  
// When the `wee_alloc` feature is enabled, use `wee_alloc` as the global  
// allocator.  
#[cfg(feature = "wee_alloc")]  
#[global_allocator]  
static ALLOC: wee_alloc::WeeAlloc = wee_alloc::WeeAlloc::INIT;  
  
#[wasm_bindgen]  
extern {  
    fn alert(s: &str);  
}  
  
#[wasm_bindgen]  
pub fn greet() {  
    alert("Hello, wasm-game-of-life!");  
}
```

wasm-game-of-life/src/utils.rs

The `src/utils.rs` module provides common utilities to make working with Rust compiled to WebAssembly easier. We will take a look at some of these utilities in more detail later in the tutorial, such as when we look at [debugging our wasm code](#), but we can ignore this file for now.

Build the Project

We use `wasm-pack` to orchestrate the following build steps:

- Ensure that we have Rust 1.30 or newer and the `wasm32-unknown-unknown` target installed via `rustup`,
- Compile our Rust sources into a WebAssembly `.wasm` binary via `cargo`,
- Use `wasm-bindgen` to generate the JavaScript API for using our Rust-generated WebAssembly.

To do all of that, run this command inside the project directory:

```
wasm-pack build
```

When the build has completed, we can find its artifacts in the `pkg` directory, and it should have these contents:

```
pkg/  
├── package.json  
├── README.md  
├── wasm_game_of_life_bg.wasm  
├── wasm_game_of_life.d.ts  
└── wasm_game_of_life.js
```

The `README.md` file is copied from the main project, but the others are completely new.

```
wasm-game-of-life/pkg/wasm_game_of_life_bg.wasm
```

The `.wasm` file is the WebAssembly binary that is generated by the Rust compiler from our Rust sources. It contains the compiled-to-wasm versions of all of our Rust functions and data. For example, it has an exported "greet" function.

```
wasm-game-of-life/pkg/wasm_game_of_life.js
```

The `.js` file is generated by `wasm-bindgen` and contains JavaScript glue for importing DOM and JavaScript functions into Rust and exposing a nice API to the WebAssembly functions to JavaScript. For example, there is a JavaScript `greet` function that wraps the `greet` function

exported from the WebAssembly module. Right now, this glue isn't doing much, but when we start passing more interesting values back and forth between wasm and JavaScript, it will help shepherd those values across the boundary.

```
import * as wasm from './wasm_game_of_life_bg';

// ...

export function greet() {
  return wasm.greet();
}
```

wasm-game-of-life/pkg/wasm_game_of_life.d.ts

The `.d.ts` file contains [TypeScript](#) type declarations for the JavaScript glue. If you are using TypeScript, you can have your calls into WebAssembly functions type checked, and your IDE can provide autocompletions and suggestions! If you aren't using TypeScript, you can safely ignore this file.

```
export function greet(): void;
```

wasm-game-of-life/pkg/package.json

The `package.json` file contains metadata about the generated JavaScript and WebAssembly package. This is used by npm and JavaScript bundlers to determine dependencies across packages, package names, versions, and a bunch of other stuff. It helps us integrate with JavaScript tooling and allows us to publish our package to npm.

```
{
  "name": "wasm-game-of-life",
  "collaborators": [
    "Your Name <your.email@example.com>"
  ],
  "description": null,
  "version": "0.1.0",
  "license": null,
  "repository": null,
  "files": [
    "wasm_game_of_life_bg.wasm",
    "wasm_game_of_life.d.ts"
  ],
  "main": "wasm_game_of_life.js",
  "types": "wasm_game_of_life.d.ts"
}
```

Putting it into a Web Page

To take our `wasm-game-of-life` package and use it in a Web page, we use the `create-wasm-app` JavaScript project template.

Run this command within the `wasm-game-of-life` directory:

```
npm init wasm-app www
```

Here's what our new `wasm-game-of-life/www` subdirectory contains:

```
wasm-game-of-life/www/  
├── bootstrap.js  
├── index.html  
├── index.js  
├── LICENSE-APACHE  
├── LICENSE-MIT  
├── package.json  
├── README.md  
└── webpack.config.js
```

Once again, let's take a closer look at some of these files.

`wasm-game-of-life/www/package.json`

This `package.json` comes pre-configured with `webpack` and `webpack-dev-server` dependencies, as well as a dependency on `hello-wasm-pack`, which is a version of the initial `wasm-pack-template` package that has been published to npm.

`wasm-game-of-life/www/webpack.config.js`

This file configures webpack and its local development server. It comes pre-configured, and you shouldn't have to tweak this at all to get webpack and its local development server working.

`wasm-game-of-life/www/index.html`

This is the root HTML file for the Web page. It doesn't do much other than load `bootstrap.js`, which is a very thin wrapper around `index.js`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello wasm-pack!</title>
  </head>
  <body>
    <script src="./bootstrap.js"></script>
  </body>
</html>
```

`wasm-game-of-life/www/index.js`

The `index.js` is the main entry point for our Web page's JavaScript. It imports the `hello-wasm-pack` npm package, which contains the default `wasm-pack-template`'s compiled WebAssembly and JavaScript glue, then it calls `hello-wasm-pack`'s `greet` function.

```
import * as wasm from "hello-wasm-pack";

wasm.greet();
```

Install the dependencies

First, ensure that the local development server and its dependencies are installed by running `npm install` within the `wasm-game-of-life/www` subdirectory:

```
npm install
```

This command only needs to be run once, and will install the `webpack` JavaScript bundler and its development server.

Note that `webpack` is not required for working with Rust and WebAssembly, it is just the bundler and development server we've chosen for convenience here. Parcel and Rollup should also support importing WebAssembly as ECMAScript modules. You can also use Rust and WebAssembly [without a bundler](#) if you prefer!

Using our Local `wasm-game-of-life` Package in `www`

Rather than use the `hello-wasm-pack` package from npm, we want to use our local `wasm-game-of-life` package instead. This will allow us to incrementally develop our Game of Life program.

Open up `wasm-game-of-life/www/package.json` and next to `"devDependencies"`, add the `"dependencies"` field, including a `"wasm-game-of-life": "file:../pkg"` entry:

```
{
  // ...
  "dependencies": {
    "wasm-game-of-life": "file:../pkg" // Add this three lines block!
  },
  "devDependencies": {
    //...
  }
}
```

Next, modify `wasm-game-of-life/www/index.js` to import `wasm-game-of-life` instead of the `hello-wasm-pack` package:

```
import * as wasm from "wasm-game-of-life";

wasm.greet();
```

Since we declared a new dependency, we need to install it:

```
npm install
```

Our Web page is now ready to be served locally!

Serving Locally

Next, open a new terminal for the development server. Running the server in a new terminal lets us leave it running in the background, and doesn't block us from running other commands in the meantime. In the new terminal, run this command from within the

`wasm-game-of-life/www` directory:

```
npm run start
```

Navigate your Web browser to <http://localhost:8080/> and you should be greeted with an alert message:

 Screenshot of the "Hello, wasm-game-of-life!" Web page alert

Anytime you make changes and want them reflected on <http://localhost:8080/>, just re-run the `wasm-pack build` command within the `wasm-game-of-life` directory.

Exercises

- Modify the `greet` function in `wasm-game-of-life/src/lib.rs` to take a `name: &str` parameter that customizes the alerted message, and pass your name to the `greet` function from inside `wasm-game-of-life/www/index.js`. Rebuild the `.wasm` binary with `wasm-pack build`, then refresh <http://localhost:8080/> in your Web browser and you should see a customized greeting!

► Answer

Rules of Conway's Game of Life

Note: If you are already familiar with Conway's Game of Life and its rules, feel free to skip to the next section!

Wikipedia gives a great description of the rules of Conway's Game of Life:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead, or "populated" or "unpopulated". Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

Consider the following initial universe:

 Initial Universe

We can calculate the next generation by considering each cell. The top left cell is dead. Rule (4) is the only transition rule that applies to dead cells. However, because the top left cell does not have exactly three live neighbors, the transition rule does not apply, and it remains dead in the next generation. The same goes for every other cell in the first row as well.

Things get interesting when we consider the top live cell, in the second row, third column. For live cells, any of the first three rules potentially applies. In this cell's case, it has only one live neighbor, and therefore rule (1) applies: this cell will die in the next generation. The same fate awaits the bottom live cell.

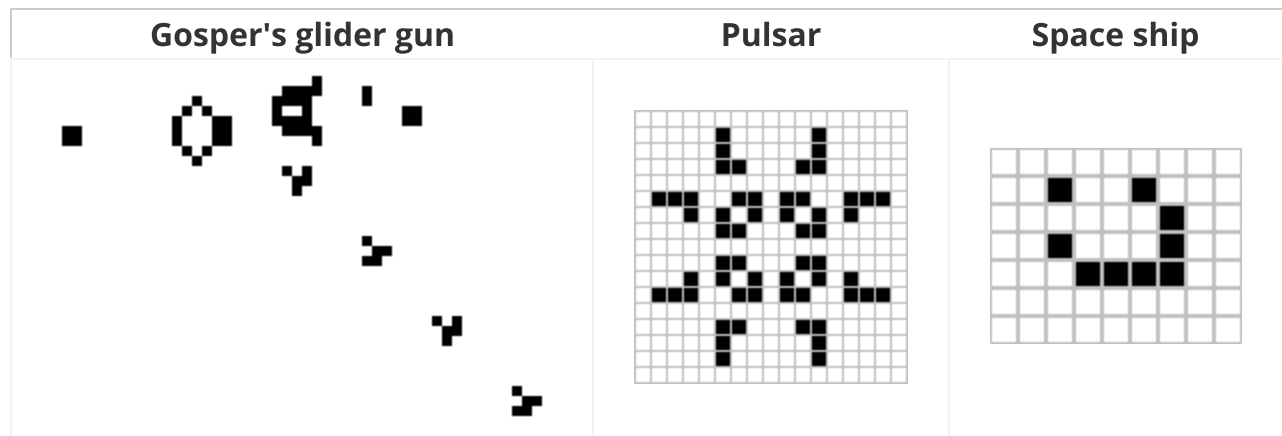
The middle live cell has two live neighbors: the top and bottom live cells. This means that rule (2) applies, and it remains live in the next generation.

The final interesting cases are the dead cells just to the left and right of the middle live cell. The three live cells are all neighbors both of these cells, which means that rule (4) applies, and these cells will become alive in the next generation.

Put it all together, and we get this universe after the next tick:

 Next Universe

From these simple, deterministic rules, strange and exciting behavior emerges:



epic conway's game of life



Exercises

- Compute by hand the next tick of our example universe. Notice anything familiar?
▶ Answer
- Can you find an initial universe that is stable? That is, a universe in which every generation is always the same.
▶ Answer

Implementing Conway's Game of Life

Design

Before we dive in, we have some design choices to consider.

Infinite Universe

The Game of Life is played in an infinite universe, but we do not have infinite memory and compute power. Working around this rather annoying limitation usually comes in one of three flavors:

1. Keep track of which subset of the universe has interesting things happening, and expand this region as needed. In the worst case, this expansion is unbounded and the implementation will get slower and slower and eventually run out of memory.
2. Create a fixed-size universe, where cells on the edges have fewer neighbors than cells in the middle. The downside with this approach is that infinite patterns, like gliders, that reach the end of the universe are snuffed out.
3. Create a fixed-size, periodic universe, where cells on the edges have neighbors that wrap around to the other side of the universe. Because neighbors wrap around the edges of the universe, gliders can keep running forever.

We will implement the third option.

Interfacing Rust and JavaScript

⚡ This is one of the most important concepts to understand and take away from this tutorial!

JavaScript's garbage-collected heap — where `Object`s, `Array`s, and DOM nodes are allocated — is distinct from WebAssembly's linear memory space, where our Rust values live. WebAssembly currently has no direct access to the garbage-collected heap (as of April 2018, this is expected to change with the ["Interface Types" proposal](#)). JavaScript, on the other hand, can read and write to the WebAssembly linear memory space, but only as an `ArrayBuffer` of scalar values (`u8`, `i32`, `f64`, etc...). WebAssembly functions also take and return scalar values. These are the building blocks from which all WebAssembly and JavaScript communication is constituted.

`wasm_bindgen` defines a common understanding of how to work with compound structures across this boundary. It involves boxing Rust structures, and wrapping the pointer in a JavaScript class for usability, or indexing into a table of JavaScript objects from Rust. `wasm_bindgen` is very convenient, but it does not remove the need to consider our data representation, and what values and structures are passed across this boundary. Instead, think of it as a tool for implementing the interface design you choose.

When designing an interface between WebAssembly and JavaScript, we want to optimize for the following properties:

1. **Minimizing copying into and out of the WebAssembly linear memory.** Unnecessary copies impose unnecessary overhead.
2. **Minimizing serializing and deserializing.** Similar to copies, serializing and deserializing also imposes overhead, and often imposes copying as well. If we can pass opaque

handles to a data structure — instead of serializing it on one side, copying it into some known location in the WebAssembly linear memory, and deserializing on the other side — we can often reduce a lot of overhead. `wasm_bindgen` helps us define and work with opaque handles to JavaScript `Object`s or boxed Rust structures.


As a general rule of thumb, a good JavaScript↔WebAssembly interface design is often one where large, long-lived data structures are implemented as Rust types that live in the WebAssembly linear memory, and are exposed to JavaScript as opaque handles. JavaScript calls exported WebAssembly functions that take these opaque handles, transform their data, perform heavy computations, query the data, and ultimately return a small, copy-able result. By only returning the small result of the computation, we avoid copying and/or serializing everything back and forth between the JavaScript garbage-collected heap and the WebAssembly linear memory.

Interfacing Rust and JavaScript in our Game of Life

Let's start by enumerating some hazards to avoid. We don't want to copy the whole universe into and out of the WebAssembly linear memory on every tick. We do not want to allocate objects for every cell in the universe, nor do we want to impose a cross-boundary call to read and write each cell.

Where does this leave us? We can represent the universe as a flat array that lives in the WebAssembly linear memory, and has a byte for each cell. `0` is a dead cell and `1` is a live cell.

Here is what a 4 by 4 universe looks like in memory:

 Screenshot of a 4 by 4 universe

To find the array index of the cell at a given row and column in the universe, we can use this formula:

```
index(row, column, universe) = row * width(universe) + column
```

We have several ways of exposing the universe's cells to JavaScript. To begin, we will implement `std::fmt::Display` for `Universe`, which we can use to generate a Rust `String` of the cells rendered as text characters. This Rust String is then copied from the WebAssembly linear memory into a JavaScript String in the JavaScript's garbage-collected heap, and is then displayed by setting HTML `textContent`. Later in the chapter, we'll evolve this implementation to avoid copying the universe's cells between heaps and to render to `<canvas>`.

Another viable design alternative would be for Rust to return a list of every cell that changed states after each tick, instead of exposing the whole universe to JavaScript. This way, JavaScript wouldn't need to iterate over the whole universe when rendering, only the relevant subset. The trade off is that this delta-based design is slightly more difficult to implement.

Rust Implementation

In the last chapter, we cloned an initial project template. We will modify that project template now.

Let's begin by removing the `alert` import and `greet` function from `wasm-game-of-life/src/lib.rs`, and replacing them with a type definition for cells:

```
#[wasm_bindgen]
#[repr(u8)]
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Cell {
    Dead = 0,
    Alive = 1,
}
```

It is important that we have `#[repr(u8)]`, so that each cell is represented as a single byte. It is also important that the `Dead` variant is `0` and that the `Alive` variant is `1`, so that we can easily count a cell's live neighbors with addition.

Next, let's define the universe. The universe has a width and a height, and a vector of cells of length `width * height`.

```
#[wasm_bindgen]
pub struct Universe {
    width: u32,
    height: u32,
    cells: Vec<Cell>,
}
```

To access the cell at a given row and column, we translate the row and column into an index into the cells vector, as described earlier:

```
impl Universe {
    fn get_index(&self, row: u32, column: u32) -> usize {
        (row * self.width + column) as usize
    }

    // ...
}
```

In order to calculate the next state of a cell, we need to get a count of how many of its neighbors are alive. Let's write a `live_neighbor_count` method to do just that!

```

impl Universe {
    // ...

    fn live_neighbor_count(&self, row: u32, column: u32) -> u8 {
        let mut count = 0;
        for delta_row in [self.height - 1, 0, 1].iter().cloned() {
            for delta_col in [self.width - 1, 0, 1].iter().cloned() {
                if delta_row == 0 && delta_col == 0 {
                    continue;
                }

                let neighbor_row = (row + delta_row) % self.height;
                let neighbor_col = (column + delta_col) % self.width;
                let idx = self.get_index(neighbor_row, neighbor_col);
                count += self.cells[idx] as u8;
            }
        }
        count
    }
}

```

The `live_neighbor_count` method uses deltas and modulo to avoid special casing the edges of the universe with `if` s. When applying a delta of `-1`, we *add* `self.height - 1` and let the modulo do its thing, rather than attempting to subtract `1`. `row` and `column` can be `0`, and if we attempted to subtract `1` from them, there would be an unsigned integer underflow.

Now we have everything we need to compute the next generation from the current one! Each of the Game's rules follows a straightforward translation into a condition on a `match` expression. Additionally, because we want JavaScript to control when ticks happen, we will put this method inside a `#[wasm_bindgen]` block, so that it gets exposed to JavaScript.

```

/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    pub fn tick(&mut self) {
        let mut next = self.cells.clone();

        for row in 0..self.height {
            for col in 0..self.width {
                let idx = self.get_index(row, col);
                let cell = self.cells[idx];
                let live_neighbors = self.live_neighbor_count(row, col);



                let next_cell = match (cell, live_neighbors) {
                    // Rule 1: Any live cell with fewer than two live neighbours
                    // dies, as if caused by underpopulation.
                    (Cell::Alive, x) if x < 2 => Cell::Dead,
                    // Rule 2: Any live cell with two or three live neighbours
                    // lives on to the next generation.
                    (Cell::Alive, 2) | (Cell::Alive, 3) => Cell::Alive,
                    // Rule 3: Any live cell with more than three live
                    // neighbours dies, as if by overpopulation.
                    (Cell::Alive, x) if x > 3 => Cell::Dead,
                    // Rule 4: Any dead cell with exactly three live neighbours
                    // becomes a live cell, as if by reproduction.
                    (Cell::Dead, 3) => Cell::Alive,
                    // All other cells remain in the same state.
                    (otherwise, _) => otherwise,
                };

                next[idx] = next_cell;
            }
        }

        self.cells = next;
    }

    // ...
}

```

So far, the state of the universe is represented as a vector of cells. To make this human readable, let's implement a basic text renderer. The idea is to write the universe line by line as text, and for each cell that is alive, print the Unicode character  ("black medium square"). For dead cells, we'll print  (a "white medium square").

By implementing the `Display` trait from Rust's standard library, we can add a way to format a structure in a user-facing manner. This will also automatically give us a `to_string` method.

```

use std::fmt;

impl fmt::Display for Universe {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        for line in self.cells.as_slice().chunks(self.width as usize) {
            for &cell in line {
                let symbol = if cell == Cell::Dead { '□' } else { '■' };
                write!(f, "{}", symbol)?;
            }
            write!(f, "\n")?;
        }

        Ok(())
    }
}

```

Finally, we define a constructor that initializes the universe with an interesting pattern of live and dead cells, as well as a `render` method:

```

/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    // ...

    pub fn new() -> Universe {
        let width = 64;
        let height = 64;

        let cells = (0..width * height)
            .map(|i| {
                if i % 2 == 0 || i % 7 == 0 {
                    Cell::Alive
                } else {
                    Cell::Dead
                }
            })
            .collect();

        Universe {
            width,
            height,
            cells,
        }
    }

    pub fn render(&self) -> String {
        self.to_string()
    }
}

```

With that, the Rust half of our Game of Life implementation is complete!

Recompile it to WebAssembly by running `wasm-pack build` within the `wasm-game-of-life` directory.

Rendering with JavaScript

First, let's add a `<pre>` element to `wasm-game-of-life/www/index.html` to render the universe into, just above the `<script>` tag:

```
<body>
  <pre id="game-of-life-canvas"></pre>
  <script src="./bootstrap.js"></script>
</body>
```

Additionally, we want the `<pre>` centered in the middle of the Web page. We can use CSS flex boxes to accomplish this task. Add the following `<style>` tag inside

`wasm-game-of-life/www/index.html` 's `<head>`:

```
<style>
  body {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
  }
</style>
```

At the top of `wasm-game-of-life/www/index.js`, let's fix our import to bring in the `Universe` rather than the old `greet` function:

```
import { Universe } from "wasm-game-of-life";
```

Also, let's get that `<pre>` element we just added and instantiate a new universe:

```
const pre = document.getElementById("game-of-life-canvas");
const universe = Universe.new();
```

The JavaScript runs in a `requestAnimationFrame` loop. On each iteration, it draws the current universe to the `<pre>`, and then calls `Universe::tick`.

```
const renderLoop = () => {  
  pre.textContent = universe.render();  
  universe.tick();  
  
  requestAnimationFrame(renderLoop);  
};
```

To start the rendering process, all we have to do is make the initial call for the first iteration of the rendering loop:

```
requestAnimationFrame(renderLoop);
```

Make sure your development server is still running (run `npm run start` inside `wasm-game-of-life/www`) and this is what <http://localhost:8080/> should look like:

 Screenshot of the Game of Life implementation with text rendering

Rendering to Canvas Directly from Memory

Generating (and allocating) a `String` in Rust and then having `wasm-bindgen` convert it to a valid JavaScript string makes unnecessary copies of the universe's cells. As the JavaScript code already knows the width and height of the universe, and can read WebAssembly's linear memory that make up the cells directly, we'll modify the `render` method to return a pointer to the start of the cells array.

Also, instead of rendering Unicode text, we'll switch to using the [Canvas API](#). We will use this design in the rest of the tutorial.

Inside `wasm-game-of-life/www/index.html`, let's replace the `<pre>` we added earlier with a `<canvas>` we will render into (it too should be within the `<body>`, before the `<script>` that loads our JavaScript):

```
<body>  
  <canvas id="game-of-life-canvas"></canvas>  
  <script src='./bootstrap.js'></script>  
</body>
```

To get the necessary information from the Rust implementation, we'll need to add some more getter functions for a universe's width, height, and pointer to its cells array. All of these are exposed to JavaScript as well. Make these additions to `wasm-game-of-life/src/lib.rs`:

```

/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    // ...

    pub fn width(&self) -> u32 {
        self.width
    }

    pub fn height(&self) -> u32 {
        self.height
    }

    pub fn cells(&self) -> *const Cell {
        self.cells.as_ptr()
    }
}

```

Next, in `wasm-game-of-life/www/index.js`, let's also import `Cell` from `wasm-game-of-life`, and define some constants that we will use when rendering to the canvas:

```

import { Universe, Cell } from "wasm-game-of-life";

const CELL_SIZE = 5; // px
const GRID_COLOR = "#CCCCCC";
const DEAD_COLOR = "#FFFFFF";
const ALIVE_COLOR = "#000000";

```

Now, let's rewrite the rest of this JavaScript code to no longer write to the `<pre>`'s `textContent` but instead draw to the `<canvas>`:

```

// Construct the universe, and get its width and height.
const universe = Universe.new();
const width = universe.width();
const height = universe.height();

// Give the canvas room for all of our cells and a 1px border
// around each of them.
const canvas = document.getElementById("game-of-life-canvas");
canvas.height = (CELL_SIZE + 1) * height + 1;
canvas.width = (CELL_SIZE + 1) * width + 1;

const ctx = canvas.getContext('2d');

const renderLoop = () => {
    universe.tick();

    drawGrid();
    drawCells();

    requestAnimationFrame(renderLoop);
};

```


To draw the grid between cells, we draw a set of equally-spaced horizontal lines, and a set of equally-spaced vertical lines. These lines criss-cross to form the grid.

```
const drawGrid = () => {
  ctx.beginPath();
  ctx.strokeStyle = GRID_COLOR;

  // Vertical lines.
  for (let i = 0; i <= width; i++) {
    ctx.moveTo(i * (CELL_SIZE + 1) + 1, 0);
    ctx.lineTo(i * (CELL_SIZE + 1) + 1, (CELL_SIZE + 1) * height + 1);
  }

  // Horizontal lines.
  for (let j = 0; j <= height; j++) {
    ctx.moveTo(0, j * (CELL_SIZE + 1) + 1);
    ctx.lineTo((CELL_SIZE + 1) * width + 1, j * (CELL_SIZE + 1) + 1);
  }

  ctx.stroke();
};
```

We can directly access WebAssembly's linear memory via `memory`, which is defined in the raw wasm module `wasm_game_of_life_bg`. To draw the cells, we get a pointer to the universe's cells, construct a `Uint8Array` overlaying the cells buffer, iterate over each cell, and draw a white or black rectangle depending on whether the cell is dead or alive, respectively. By working with pointers and overlays, we avoid copying the cells across the boundary on every tick.

```
// Import the WebAssembly memory at the top of the file.
import { memory } from "wasm-game-of-life/wasm_game_of_life_bg";

// ...

const getIndex = (row, column) => {
    return row * width + column;
};

const drawCells = () => {
    const cellsPtr = universe.cells();
    const cells = new Uint8Array(memory.buffer, cellsPtr, width * height);

    ctx.beginPath();

    for (let row = 0; row < height; row++) {
        for (let col = 0; col < width; col++) {
            const idx = getIndex(row, col);

            ctx.fillStyle = cells[idx] === Cell.Dead
                ? DEAD_COLOR
                : ALIVE_COLOR;

            ctx.fillRect(
                col * (CELL_SIZE + 1) + 1,
                row * (CELL_SIZE + 1) + 1,
                CELL_SIZE,
                CELL_SIZE
            );
        }
    }

    ctx.stroke();
};
```

To start the rendering process, we'll use the same code as above to start the first iteration of the rendering loop:

```
drawGrid();
drawCells();
requestAnimationFrame(renderLoop);
```

Note that we call `drawGrid()` and `drawCells()` here *before* we call `requestAnimationFrame()`. The reason we do this is so that the *initial* state of the universe is drawn before we make modifications. If we instead simply called `requestAnimationFrame(renderLoop)`, we'd end up with a situation where the first frame that was drawn would actually be *after* the first call to `universe.tick()`, which is the second "tick" of the life of these cells.

It Works!

Rebuild the WebAssembly and bindings glue by running this command from within the root `wasm-game-of-life` directory:

```
wasm-pack build
```

Make sure your development server is still running. If it isn't, start it again from within the `wasm-game-of-life/www` directory:

```
npm run start
```

If you refresh <http://localhost:8080/>, you should be greeted with an exciting display of life!

 Screenshot of the Game of Life implementation

As an aside, there is also a really neat algorithm for implementing the Game of Life called [hashlife](#). It uses aggressive memoizing and can actually get *exponentially faster* to compute future generations the longer it runs! Given that, you might be wondering why we didn't implement hashlife in this tutorial. It is out of scope for this text, where we are focusing on Rust and WebAssembly integration, but we highly encourage you to go learn about hashlife on your own!

Exercises

- Initialize the universe with a single space ship.
- Instead of hard-coding the initial universe, generate a random one, where each cell has a fifty-fifty chance of being alive or dead.

Hint: use the `js-sys` crate to import the `Math.random` JavaScript function.

► Answer

- Representing each cell with a byte makes iterating over cells easy, but it comes at the cost of wasting memory. Each byte is eight bits, but we only require a single bit to represent whether each cell is alive or dead. Refactor the data representation so that each cell uses only a single bit of space.

► Answer

Testing Conway's Game of Life

Now that we have our Rust implementation of the Game of Life rendering in the browser with JavaScript, let's talk about testing our Rust-generated WebAssembly functions.

We are going to test our `tick` function to make sure that it gives us the output that we expect.

Next, we'll want to create some setter and getter functions inside our existing `impl Universe` block in the `wasm_game_of_life/src/lib.rs` file. We are going to create a `set_width` and a `set_height` function so we can create `Universe`s of different sizes.

```
#[wasm_bindgen]
impl Universe {
    // ...

    /// Set the width of the universe.
    ///
    /// Resets all cells to the dead state.
    pub fn set_width(&mut self, width: u32) {
        self.width = width;
        self.cells = (0..width * self.height).map(|_i| Cell::Dead).collect();
    }

    /// Set the height of the universe.
    ///
    /// Resets all cells to the dead state.
    pub fn set_height(&mut self, height: u32) {
        self.height = height;
        self.cells = (0..self.width * height).map(|_i| Cell::Dead).collect();
    }
}
```

We are going to create another `impl Universe` block inside our `wasm_game_of_life/src/lib.rs` file without the `#[wasm_bindgen]` attribute. There are a few functions we need for testing that we don't want to expose to our JavaScript. Rust-generated WebAssembly functions cannot return borrowed references. Try compiling the Rust-generated WebAssembly with the attribute and take a look at the errors you get.

We are going to write the implementation of `get_cells` to get the contents of the `cells` of a `Universe`. We'll also write a `set_cells` function so we can set `cells` in a specific row and column of a `Universe` to be `Alive`.

```
impl Universe {
    /// Get the dead and alive values of the entire universe.
    pub fn get_cells(&self) -> &[Cell] {
        &self.cells
    }

    /// Set cells to be alive in a universe by passing the row and column
    /// of each cell as an array.
    pub fn set_cells(&mut self, cells: &[(u32, u32)]) {
        for (row, col) in cells.iter().cloned() {
            let idx = self.get_index(row, col);
            self.cells[idx] = Cell::Alive;
        }
    }
}
```

Now we're going to create our test in the `wasm_game_of_life/tests/web.rs` file.

Before we do that, there is already one working test in the file. You can confirm that the Rust-generated WebAssembly test is working by running `wasm-pack test --chrome --headless` in the `wasm-game-of-life` directory. You can also use the `--firefox`, `--safari`, and `--node` options to test your code in those browsers.

In the `wasm_game_of_life/tests/web.rs` file, we need to export our `wasm_game_of_life` crate and the `Universe` type.

```
extern crate wasm_game_of_life;
use wasm_game_of_life::Universe;
```

In the `wasm_game_of_life/tests/web.rs` file we'll want to create some spaceship builder functions.

We'll want one for our input spaceship that we'll call the `tick` function on and we'll want the expected spaceship we will get after one tick. We picked the cells that we want to initialize as `Alive` to create our spaceship in the `input_spaceship` function. The position of the spaceship in the `expected_spaceship` function after the tick of the `input_spaceship` was calculated manually. You can confirm for yourself that the cells of the input spaceship after one tick is the same as the expected spaceship.

```
#[cfg(test)]
pub fn input_spaceship() -> Universe {
    let mut universe = Universe::new();
    universe.set_width(6);
    universe.set_height(6);
    universe.set_cells(&[(1,2), (2,3), (3,1), (3,2), (3,3)]);
    universe
}

#[cfg(test)]
pub fn expected_spaceship() -> Universe {
    let mut universe = Universe::new();
    universe.set_width(6);
    universe.set_height(6);
    universe.set_cells(&[(2,1), (2,3), (3,2), (3,3), (4,2)]);
    universe
}
```

Now we will write the implementation for our `test_tick` function. First, we create an instance of our `input_spaceship()` and our `expected_spaceship()`. Then, we call `tick` on the `input_universe`. Finally, we use the `assert_eq!` macro to call `get_cells()` to ensure that `input_universe` and `expected_universe` have the same `Cell` array values. We add the `#[wasm_bindgen_test]` attribute to our code block so we can test our Rust-generated WebAssembly code and use `wasm-pack test` to test the WebAssembly code.

```
#[wasm_bindgen_test]
pub fn test_tick() {
    // Let's create a smaller Universe with a small spaceship to test!
    let mut input_universe = input_spaceship();

    // This is what our spaceship should look like
    // after one tick in our universe.
    let expected_universe = expected_spaceship();

    // Call `tick` and then see if the cells in the `Universe`s are the same.
    input_universe.tick();
    assert_eq!(&input_universe.get_cells(), &expected_universe.get_cells());
}
```

Run the tests within the `wasm-game-of-life` directory by running `wasm-pack test --firefox --headless`.

Debugging

Before we write much more code, we will want to have some debugging tools in our belt for when things go wrong. Take a moment to review the [reference page listing tools and approaches available for debugging Rust-generated WebAssembly](#).

Enable Logging for Panics

If our code panics, we want informative error messages to appear in the developer console.

Our `wasm-pack-template` comes with an optional, enabled-by-default dependency on the `console_error_panic_hook` crate that is configured in `wasm-game-of-life/src/utils.rs`. All we need to do is install the hook in an initialization function or common code path. We can call it inside the `Universe::new` constructor in `wasm-game-of-life/src/lib.rs`:

```
pub fn new() -> Universe {
    utils::set_panic_hook();

    // ...
}
```

Add Logging to our Game of Life

Let's use the `console.log` function via the `web-sys` crate to add some logging about each cell in our `Universe::tick` function.

First, add `web-sys` as a dependency and enable its `"console"` feature in `wasm-game-of-life/Cargo.toml`:

```
[dependencies]

# ...

[dependencies.web-sys]
version = "0.3"
features = [
    "console",
]
```

For ergonomics, we'll wrap the `console.log` function up in a `println!`-style macro:

```
extern crate web_sys;

// A macro to provide `println!(..)`-style syntax for `console.log` logging.
macro_rules! log {
    ( $( $t:tt )* ) => {
        web_sys::console::log_1(&format!( $( $t )* ).into());
    }
}
```

Now, we can start logging messages to the console by inserting calls to `log` in Rust code. For example, to log each cell's state, live neighbors count, and next state, we could modify

`wasm-game-of-life/src/lib.rs` like this:

```
diff --git a/src/lib.rs b/src/lib.rs
index f757641..a30e107 100755
--- a/src/lib.rs
+++ b/src/lib.rs
@@ -123,6 +122,14 @@ impl Universe {
    let cell = self.cells[idx];
    let live_neighbors = self.live_neighbor_count(row, col);

+    log!(
+        "cell[{}, {}] is initially {:?} and has {} live neighbors",
+        row,
+        col,
+        cell,
+        live_neighbors
+    );
+
    let next_cell = match (cell, live_neighbors) {
        // Rule 1: Any live cell with fewer than two live neighbours
        // dies, as if caused by underpopulation.
@@ -140,6 +147,8 @@ impl Universe {
        (otherwise, _) => otherwise,
    };

+    log!("    it becomes {:?}", next_cell);
+
    next[idx] = next_cell;
}
}
```

Using a Debugger to Pause Between Each Tick

Browser's stepping debuggers are useful for inspecting the JavaScript that our Rust-generated WebAssembly interacts with.

For example, we can use the debugger to pause on each iteration of our `renderLoop` function by placing a JavaScript `debugger;` statement above our call to `universe.tick()`.

```
const renderLoop = () => {
    debugger;
    universe.tick();

    drawGrid();
    drawCells();

    requestAnimationFrame(renderLoop);
};
```


This provides us with a convenient checkpoint for inspecting logged messages, and comparing the currently rendered frame to the previous one.

 Screenshot of debugging the Game of Life

Exercises

- Add logging to the `tick` function that records the row and column of each cell that transitioned states from live to dead or vice versa.
- Introduce a `panic!()` in the `Universe::new` method. Inspect the panic's backtrace in your Web browser's JavaScript debugger. Disable debug symbols, rebuild without the `console_error_panic_hook` optional dependency, and inspect the stack trace again. Not as useful is it?

Adding Interactivity

We will continue to explore the JavaScript and WebAssembly interface by adding some interactive features to our Game of Life implementation. We will enable users to toggle whether a cell is alive or dead by clicking on it, and allow pausing the game, which makes drawing cell patterns a lot easier.

Pausing and Resuming the Game

Let's add a button to toggle whether the game is playing or paused. To

`wasm-game-of-life/www/index.html`, add the button right above the `<canvas>`:

```
<button id="play-pause"></button>
```

In the `wasm-game-of-life/www/index.js` JavaScript, we will make the following changes:

- Keep track of the identifier returned by the latest call to `requestAnimationFrame`, so that we can cancel the animation by calling `cancelAnimationFrame` with that identifier.
- When the play/pause button is clicked, check for whether we have the identifier for a queued animation frame. If we do, then the game is currently playing, and we want to cancel the animation frame so that `renderLoop` isn't called again, effectively pausing the game. If we do not have an identifier for a queued animation frame, then we are currently paused, and we would like to call `requestAnimationFrame` to resume the game.

Because the JavaScript is driving the Rust and WebAssembly, this is all we need to do, and we don't need to change the Rust sources.

We introduce the `animationId` variable to keep track of the identifier returned by `requestAnimationFrame`. When there is no queued animation frame, we set this variable to `null`.

```
let animationId = null;

// This function is the same as before, except the
// result of `requestAnimationFrame` is assigned to
// `animationId`.
const renderLoop = () => {
  drawGrid();
  drawCells();

  universe.tick();

  animationId = requestAnimationFrame(renderLoop);
};
```

At any instant in time, we can tell whether the game is paused or not by inspecting the value of `animationId`:

```
const isPaused = () => {
  return animationId === null;
};
```

Now, when the play/pause button is clicked, we check whether the game is currently paused or playing, and resume the `renderLoop` animation or cancel the next animation frame respectively. Additionally, we update the button's text icon to reflect the action that the button will take when clicked next.

```

const playPauseButton = document.getElementById("play-pause");

const play = () => {
  playPauseButton.textContent = "⏮";
  renderLoop();
};

const pause = () => {
  playPauseButton.textContent = "▶";
  cancelAnimationFrame(animationId);
  animationId = null;
};

playPauseButton.addEventListener("click", event => {
  if (isPaused()) {
    play();
  } else {
    pause();
  }
});

```

Finally, we were previously kick-starting the game and its animation by calling `requestAnimationFrame(renderLoop)` directly, but we want to replace that with a call to `play` so that the button gets the correct initial text icon.

```

// This used to be `requestAnimationFrame(renderLoop)`.
play();

```

Refresh <http://localhost:8080/> and we should now be able to pause and resume the game by clicking on the button!

Toggling a Cell's State on "click" Events

Now that we can pause the game, it's time to add the ability to mutate the cells by clicking on them.

To toggle a cell is to flip its state from alive to dead or from dead to alive. Add a `toggle` method to `Cell` in `wasm-game-of-life/src/lib.rs`:

```

impl Cell {
  fn toggle(&mut self) {
    *self = match *self {
      Cell::Dead => Cell::Alive,
      Cell::Alive => Cell::Dead,
    };
  }
}

```

To toggle the state of a cell at given row and column, we translate the row and column pair into an index into the cells vector and call the toggle method on the cell at that index:

```
/// Public methods, exported to JavaScript.
#[wasm_bindgen]
impl Universe {
    // ...

    pub fn toggle_cell(&mut self, row: u32, column: u32) {
        let idx = self.get_index(row, column);
        self.cells[idx].toggle();
    }
}
```

This method is defined within the `impl` block that is annotated with `#[wasm_bindgen]` so that it can be called by JavaScript.

In `wasm-game-of-life/www/index.js`, we listen to click events on the `<canvas>` element, translate the click event's page-relative coordinates into canvas-relative coordinates, and then into a row and column, invoke the `toggle_cell` method, and finally redraw the scene.

```
canvas.addEventListener("click", event => {
    const boundingRect = canvas.getBoundingClientRect();

    const scaleX = canvas.width / boundingRect.width;
    const scaleY = canvas.height / boundingRect.height;

    const canvasLeft = (event.clientX - boundingRect.left) * scaleX;
    const canvasTop = (event.clientY - boundingRect.top) * scaleY;

    const row = Math.min(Math.floor(canvasTop / (CELL_SIZE + 1)), height - 1);
    const col = Math.min(Math.floor(canvasLeft / (CELL_SIZE + 1)), width - 1);

    universe.toggle_cell(row, col);

    drawGrid();
    drawCells();
});
```

Rebuild with `wasm-pack build` in `wasm-game-of-life`, then refresh <http://localhost:8080/> again and we can now draw our own patterns by clicking on the cells and toggling their state.

Exercises

- Introduce an `<input type="range">` widget to control how many ticks occur per animation frame.

- Add a button that resets the universe to a random initial state when clicked. Another button that resets the universe to all dead cells.
- On `Ctrl + Click`, insert a `glider` centered on the target cell. On `Shift + Click`, insert a pulsar.

Time Profiling

In this chapter, we will improve the performance of our Game of Life implementation. We will use time profiling to guide our efforts.

Familiarize yourself with [the available tools for time profiling Rust and WebAssembly code](#) before continuing.

Creating a Frames Per Second Timer with the `window.performance.now` Function

This FPS timer will be useful as we investigate speeding up our Game of Life's rendering.

We start by adding an `fps` object to `wasm-game-of-life/www/index.js`:

```

const fps = new class {
  constructor() {
    this.fps = document.getElementById("fps");
    this.frames = [];
    this.lastFrameTimeStamp = performance.now();
  }

  render() {
    // Convert the delta time since the last frame render into a measure
    // of frames per second.
    const now = performance.now();
    const delta = now - this.lastFrameTimeStamp;
    this.lastFrameTimeStamp = now;
    const fps = 1 / delta * 1000;

    // Save only the latest 100 timings.
    this.frames.push(fps);
    if (this.frames.length > 100) {
      this.frames.shift();
    }

    // Find the max, min, and mean of our 100 latest timings.
    let min = Infinity;
    let max = -Infinity;
    let sum = 0;
    for (let i = 0; i < this.frames.length; i++) {
      sum += this.frames[i];
      min = Math.min(this.frames[i], min);
      max = Math.max(this.frames[i], max);
    }
    let mean = sum / this.frames.length;

    // Render the statistics.
    this.fps.textContent = `
Frames per Second:
    latest = ${Math.round(fps)}
avg of last 100 = ${Math.round(mean)}
min of last 100 = ${Math.round(min)}
max of last 100 = ${Math.round(max)}
`.trim();
  }
};

```

Next we call the `fps.render` function on each iteration of `renderLoop`:

```

const renderLoop = () => {
  fps.render(); //new

  universe.tick();
  drawGrid();
  drawCells();

  animationId = requestAnimationFrame(renderLoop);
};

```

Finally, don't forget to add the `fps` element to `wasm-game-of-life/www/index.html`, just above the `<canvas>`:

```
<div id="fps"></div>
```

And add CSS to make its formatting nice:

```
#fps {  
  white-space: pre;  
  font-family: monospace;  
}
```

And voila! Refresh <http://localhost:8080> and now we have an FPS counter!

Time Each `Universe::tick` with `console.time` and `console.timeEnd`

To measure how long each invocation of `Universe::tick` takes, we can use `console.time` and `console.timeEnd` via the `web-sys` crate.

First, add `web-sys` as a dependency to `wasm-game-of-life/Cargo.toml`:

```
[dependencies.web-sys]  
version = "0.3"  
features = [  
  "console",  
]
```

Because there should be a corresponding `console.timeEnd` invocation for every `console.time` call, it is convenient to wrap them both up in an [RAII](#) type:

```
extern crate web_sys;
use web_sys::console;

pub struct Timer<'a> {
    name: &'a str,
}

impl<'a> Timer<'a> {
    pub fn new(name: &'a str) -> Timer<'a> {
        console::time_with_label(name);
        Timer { name }
    }
}

impl<'a> Drop for Timer<'a> {
    fn drop(&mut self) {
        console::time_end_with_label(self.name);
    }
}
```

Then, we can time how long each `Universe::tick` takes by adding this snippet to the top of the method:

```
let _timer = Timer::new("Universe::tick");
```


The time of how long each call to `Universe::tick` took are now logged in the console:

 Screenshot of console.time logs

Additionally, `console.time` and `console.timeEnd` pairs will show up in your browser's profiler's "timeline" or "waterfall" view:


 Screenshot of console.time logs

Growing our Game of Life Universe

 This section utilizes example screenshots from Firefox. While all modern browsers have similar tools, there might be slight nuances to working with different developer tools. The profile information you extract will be essentially the same, but your mileage might vary in terms of the views you see and the naming of different tools.

What happens if we make our Game of Life universe larger? Replacing the 64 by 64 universe with a 128 by 128 universe (by modifying `Universe::new` in `wasm-game-of-life/src/lib.rs`) results in FPS dropping from a smooth 60 to a choppy 40-ish on my machine.

If we record a profile and look at the waterfall view, we see that each animation frame is taking over 20 milliseconds. Recall that 60 frames per second leaves sixteen milliseconds for the whole process of rendering a frame. That's not just our JavaScript and WebAssembly, but also everything else the browser is doing, such as painting.

 Screenshot of a waterfall view of rendering a frame

If we look at what happens within a single animation frame, we see that the `CanvasRenderingContext2D.fillStyle` setter is very expensive!

⚠ In Firefox, if you see a line that simply says "DOM" instead of the `CanvasRenderingContext2D.fillStyle` mentioned above, you may need to turn on the option for "Show Gecko Platform Data" in your performance developer tools options:

 Turning on Show Gecko Platform Data

 Screenshot of a flamegraph view of rendering a frame

And we can confirm that this isn't an abnormality by looking at the call tree's aggregation of many frames:

 Screenshot of a flamegraph view of rendering a frame

Nearly 40% of our time is spent in this setter!

⚡ We might have expected something in the `tick` method to be the performance bottleneck, but it wasn't. Always let profiling guide your focus, since time may be spent in places you don't expect it to be.

In the `drawCells` function in `wasm-game-of-life/www/index.js`, the `fillStyle` property is set once for every cell in the universe, on every animation frame:

```
for (let row = 0; row < height; row++) {  
  for (let col = 0; col < width; col++) {  
    const idx = getIndex(row, col);  
  
    ctx.fillStyle = cells[idx] === DEAD  
      ? DEAD_COLOR  
      : ALIVE_COLOR;  
  
    ctx.fillRect(  
      col * (CELL_SIZE + 1) + 1,  
      row * (CELL_SIZE + 1) + 1,  
      CELL_SIZE,  
      CELL_SIZE  
    );  
  }  
}
```

Now that we have discovered that setting `fillStyle` is so expensive, what can we do to avoid setting it so often? We need to change `fillStyle` depending on whether a cell is alive or dead. If we set `fillStyle = ALIVE_COLOR` and then draw every alive cell in one pass, and then set `fillStyle = DEAD_COLOR` and draw every dead cell in another pass, then we only end setting `fillStyle` twice, rather than once for every cell.

```
// Alive cells.
ctx.fillStyle = ALIVE_COLOR;
for (let row = 0; row < height; row++) {
  for (let col = 0; col < width; col++) {
    const idx = getIndex(row, col);
    if (cells[idx] !== Cell.Alive) {
      continue;
    }

    ctx.fillRect(
      col * (CELL_SIZE + 1) + 1,
      row * (CELL_SIZE + 1) + 1,
      CELL_SIZE,
      CELL_SIZE
    );
  }
}

// Dead cells.
ctx.fillStyle = DEAD_COLOR;
for (let row = 0; row < height; row++) {
  for (let col = 0; col < width; col++) {
    const idx = getIndex(row, col);
    if (cells[idx] !== Cell.Dead) {
      continue;
    }

    ctx.fillRect(
      col * (CELL_SIZE + 1) + 1,
      row * (CELL_SIZE + 1) + 1,
      CELL_SIZE,
      CELL_SIZE
    );
  }
}
}
```

After saving these changes and refreshing <http://localhost:8080/>, rendering is back to a smooth 60 frames per second.

If we take another profile, we can see that only about ten milliseconds are spent in each animation frame now.

 Screenshot of a waterfall view of rendering a frame after the drawCells changes

Breaking down a single frame, we see that the `fillStyle` cost is gone, and most of our frame's time is spent within `fillRect`, drawing each cell's rectangle.

 Screenshot of a flamegraph view of rendering a frame after the drawCells changes

Making Time Run Faster

Some folks don't like waiting around, and would prefer if instead of one tick of the universe occurred per animation frame, nine ticks did. We can modify the `renderLoop` function in `wasm-game-of-life/www/index.js` to do this quite easily:

```
for (let i = 0; i < 9; i++) {  
  universe.tick();  
}
```

On my machine, this brings us back down to only 35 frames per second. No good. We want that buttery 60!

Now we know that time is being spent in `Universe::tick`, so let's add some `Timer`s to wrap various bits of it in `console.time` and `console.timeEnd` calls, and see where that leads us. My hypothesis is that allocating a new vector of cells and freeing the old vector on every tick is costly, and taking up a significant portion of our time budget.

```

pub fn tick(&mut self) {
    let _timer = Timer::new("Universe::tick");

    let mut next = {
        let _timer = Timer::new("allocate next cells");
        self.cells.clone()
    };

    {
        let _timer = Timer::new("new generation");
        for row in 0..self.height {
            for col in 0..self.width {
                let idx = self.get_index(row, col);
                let cell = self.cells[idx];
                let live_neighbors = self.live_neighbor_count(row, col);

                let next_cell = match (cell, live_neighbors) {
                    // Rule 1: Any live cell with fewer than two live neighbours
                    // dies, as if caused by underpopulation.
                    (Cell::Alive, x) if x < 2 => Cell::Dead,
                    // Rule 2: Any live cell with two or three live neighbours
                    // lives on to the next generation.
                    (Cell::Alive, 2) | (Cell::Alive, 3) => Cell::Alive,
                    // Rule 3: Any live cell with more than three live
                    // neighbours dies, as if by overpopulation.
                    (Cell::Alive, x) if x > 3 => Cell::Dead,
                    // Rule 4: Any dead cell with exactly three live neighbours
                    // becomes a live cell, as if by reproduction.
                    (Cell::Dead, 3) => Cell::Alive,
                    // All other cells remain in the same state.
                    (otherwise, _) => otherwise,
                };

                next[idx] = next_cell;
            }
        }

        let _timer = Timer::new("free old cells");
        self.cells = next;
    }
}

```

Looking at the timings, it is clear that my hypothesis is incorrect: the vast majority of time is spent actually calculating the next generation of cells. Allocating and freeing a vector on every tick appears to have negligible cost, surprisingly. Another reminder to always guide our efforts with profiling!

 Screenshot of a Universe::tick timer results

The next section requires the `nightly` compiler. It's required because of the `test feature gate` we're going to use for the benchmarks. Another tool we will install is `cargo benchcmp`. It's a small utility for comparing micro-benchmarks produced by `cargo bench`.

Let's write a native code `#[bench]` doing the same thing that our WebAssembly is doing, but where we can use more mature profiling tools. Here is the new

`wasm-game-of-life/benches/bench.rs`:

```
#![feature(test)]

extern crate test;
extern crate wasm_game_of_life;

#[bench]
fn universe_ticks(b: &mut test::Bencher) {
    let mut universe = wasm_game_of_life::Universe::new();

    b.iter(|| {
        universe.tick();
    });
}
```

We also have to comment out all the `#[wasm_bindgen]` annotations, and the `"cdylib"` bits from `Cargo.toml` or else building native code will fail and have link errors.

With all that in place, we can run `cargo bench | tee before.txt` to compile and run our benchmark! The `| tee before.txt` part will take the output from `cargo bench` and put in a file called `before.txt`.

```
$ cargo bench | tee before.txt
   Finished release [optimized + debuginfo] target(s) in 0.0 secs
   Running target/release/deps/wasm_game_of_life-91574dfbe2b5a124

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Running target/release/deps/bench-8474091a05cfa2d9

running 1 test
test universe_ticks ... bench:      664,421 ns/iter (+/- 51,926)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured; 0 filtered out
```

This also tells us where the binary lives, and we can run the benchmarks again, but this time under our operating system's profiler. In my case, I'm running Linux, so `perf` is the profiler I'll use:

```
$ perf record -g target/release/deps/bench-8474091a05cfa2d9 --bench
running 1 test
test universe_ticks ... bench:      635,061 ns/iter (+/- 38,764)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured; 0 filtered out

[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.178 MB perf.data (2349 samples) ]
```

Loading up the profile with `perf report` shows that all of our time is spent in `Universe::tick`, as expected:

 Screenshot of perf report

`perf` will annotate which instructions in a function time is being spent at if you press `a`:

 Screenshot of perf's instruction annotation

This tells us that 26.67% of time is being spent summing neighboring cells' values, 23.41% of time is spent getting the neighbor's column index, and another 15.42% of time is spent getting the neighbor's row index. Of these top three most expensive instructions, the second and third are both costly `div` instructions. These `div`s implement the modulo indexing logic in `Universe::live_neighbor_count`.

Recall the `live_neighbor_count` definition inside `wasm-game-of-life/src/lib.rs`:

```
fn live_neighbor_count(&self, row: u32, column: u32) -> u8 {
    let mut count = 0;
    for delta_row in [self.height - 1, 0, 1].iter().cloned() {
        for delta_col in [self.width - 1, 0, 1].iter().cloned() {
            if delta_row == 0 && delta_col == 0 {
                continue;
            }

            let neighbor_row = (row + delta_row) % self.height;
            let neighbor_col = (column + delta_col) % self.width;
            let idx = self.get_index(neighbor_row, neighbor_col);
            count += self.cells[idx] as u8;
        }
    }
    count
}
```

The reason we used modulo was to avoid cluttering up the code with `if` branches for the first or last row or column edge cases. But we are paying the cost of a `div` instruction even for the most common case, when neither `row` nor `column` is on the edge of the universe and they don't need the modulo wrapping treatment. Instead, if we use `if`s for the edge cases and unroll this loop, the branches *should* be very well-predicted by the CPU's branch predictor.

Let's rewrite `live_neighbor_count` like this:

```
fn live_neighbor_count(&self, row: u32, column: u32) -> u8 {
    let mut count = 0;

    let north = if row == 0 {
        self.height - 1
    } else {
        row - 1
    };

    let south = if row == self.height - 1 {
        0
    } else {
        row + 1
    };

    let west = if column == 0 {
        self.width - 1
    } else {
        column - 1
    };

    let east = if column == self.width - 1 {
        0
    } else {
        column + 1
    };

    let nw = self.get_index(north, west);
    count += self.cells[nw] as u8;

    let n = self.get_index(north, column);
    count += self.cells[n] as u8;

    let ne = self.get_index(north, east);
    count += self.cells[ne] as u8;

    let w = self.get_index(row, west);
    count += self.cells[w] as u8;

    let e = self.get_index(row, east);
    count += self.cells[e] as u8;

    let sw = self.get_index(south, west);
    count += self.cells[sw] as u8;

    let s = self.get_index(south, column);
    count += self.cells[s] as u8;

    let se = self.get_index(south, east);
    count += self.cells[se] as u8;

    count
}
```

Now let's run the benchmarks again! This time output it to `after.txt`.


```
$ cargo bench | tee after.txt
Compiling wasm_game_of_life v0.1.0 (file:///home/fitzgen/wasm_game_of_life)
Finished release [optimized + debuginfo] target(s) in 0.82 secs
Running target/release/deps/wasm_game_of_life-91574dfbe2b5a124

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

Running target/release/deps/bench-8474091a05cfa2d9

running 1 test
test universe_ticks ... bench:      87,258 ns/iter (+/- 14,632)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured; 0 filtered out
```

That looks a whole lot better! We can see just how much better it is with the `benchcmp` tool and the two text files we created before:

```
$ cargo benchcmp before.txt after.txt
name           before.txt ns/iter  after.txt ns/iter  diff ns/iter   diff %  speedup
universe_ticks  664,421             87,258          -577,163   -86.87%    x 7.61
```

Wow! 7.61x speed up!

WebAssembly intentionally maps closely to common hardware architectures, but we do need to make sure that this native code speed up translates into a WebAssembly speed up as well.

Let's rebuild the `.wasm` with `wasm-pack build` and refresh <http://localhost:8080/>. On my machine, the page is running at 60 frames per second again, and recording another profile with the browser's profiler reveals that each animation frame is taking about ten milliseconds.

Success!

 Screenshot of a waterfall view of rendering a frame after replacing modulus with branches

Exercises

- At this point, the next lowest hanging fruit for speeding up `Universe::tick` is removing the allocation and free. Implement double buffering of cells, where the `Universe` maintains two vectors, never frees either of them, and never allocates new buffers in `tick`.
- Implement the alternative, delta-based design from the "Implementing Life" chapter, where the Rust code returns a list of cells that changed states to JavaScript. Does this make rendering to `<canvas>` faster? Can you implement this design without allocating a new list of deltas on every tick?

- As our profiling has shown us, 2D `<canvas>` rendering is not particularly fast. Replace the 2D canvas renderer with a [WebGL](#) renderer. How much faster is the WebGL version? How large can you make the universe before WebGL rendering is a bottleneck?

Shrinking `.wasm` Size

For `.wasm` binaries that we ship to clients over the network, such as our Game of Life Web application, we want to keep an eye on code size. The smaller our `.wasm` is, the faster our page loads get, and the happier our users are.

How small can we get our Game of Life `.wasm` binary via build configuration?

Take a moment to review the build configuration options we can tweak to get smaller `.wasm` code sizes.

With the default release build configuration (without debug symbols), our WebAssembly binary is 29,410 bytes:

```
$ wc -c pkg/wasm_game_of_life_bg.wasm
29410 pkg/wasm_game_of_life_bg.wasm
```

After enabling LTO, setting `opt-level = "z"`, and running `wasm-opt -Oz`, the resulting `.wasm` binary shrinks to only 17,317 bytes:

```
$ wc -c pkg/wasm_game_of_life_bg.wasm
17317 pkg/wasm_game_of_life_bg.wasm
```

And if we compress it with `gzip` (which nearly every HTTP server does) we get down to a measly 9,045 bytes!

```
$ gzip -9 < pkg/wasm_game_of_life_bg.wasm | wc -c
9045
```

Exercises

- Use the `wasm-strip` tool to remove the panicking infrastructure functions from our Game of Life's `.wasm` binary. How many bytes does it save?

- Build our Game of Life crate with and without `wee_alloc` as its global allocator. The `rustwasm/wasm-pack-template` template that we cloned to start this project has a "wee_alloc" cargo feature that you can enable by adding it to the `default` key in the `[features]` section of `wasm-game-of-life/Cargo.toml`:

```
[features]
default = ["wee_alloc"]
```

How much size does using `wee_alloc` shave off of the `.wasm` binary?

- We only ever instantiate a single `Universe`, so rather than providing a constructor, we can export operations that manipulate a single `static mut` global instance. If this global instance also uses the double buffering technique discussed in earlier chapters, we can make those buffers also be `static mut` globals. This removes all dynamic allocation from our Game of Life implementation, and we can make it a `#![no_std]` crate that doesn't include an allocator. How much size was removed from the `.wasm` by completely removing the allocator dependency?

Publishing to npm

Now that we have a working, fast, *and* small `wasm-game-of-life` package, we can publish it to npm so other JavaScript developers can reuse it, if they ever need an off-the-shelf Game of Life implementation.

Prerequisites

First, [make sure you have an npm account](#).

Second, make sure you are logged into your account locally, by running this command:

```
wasm-pack login
```

Publishing

Make sure that the `wasm-game-of-life/pkg` build is up to date by running `wasm-pack` inside the `wasm-game-of-life` directory:

```
wasm-pack build
```

Take a moment to check out the contents of `wasm-game-of-life/pkg` now, this is what we are publishing to npm in the next step!

When you're ready, run `wasm-pack publish` to upload the package to npm:

```
wasm-pack publish
```

That's all it takes to publish to npm!

...except other folks have also done this tutorial, and therefore the `wasm-game-of-life` name is taken on npm, and that last command probably didn't work.

Open up `wasm-game-of-life/Cargo.toml` and add your username to the end of the `name` to disambiguate the package in a unique way:

```
[package]
name = "wasm-game-of-life-my-username"
```

Then, rebuild and publish again:

```
wasm-pack build
wasm-pack publish
```

This time it should work!

Reference

This section contains reference material for Rust and WebAssembly development. It is not intended to provide a narrative and be read start to finish. Instead, each subsection should stand on its own.

Crates You Should Know

This is a curated list of awesome crates you should know about for doing Rust and WebAssembly development.

You can also browse all the crates published to crates.io in the WebAssembly category.

Interacting with JavaScript and the DOM

`wasm-bindgen` | [crates.io](#) | [repository](#)

`wasm-bindgen` facilitates high-level interactions between Rust and JavaScript. It allows one to import JavaScript things into Rust and export Rust things to JavaScript.

`wasm-bindgen-futures` | [crates.io](#) | [repository](#)

`wasm-bindgen-futures` is a bridge connecting JavaScript `Promise`s and Rust `Future`s. It can convert in both directions and is useful when working with asynchronous tasks in Rust, and allows interacting with DOM events and I/O operations.

`js-sys` | [crates.io](#) | [repository](#)

Raw `wasm-bindgen` imports for all the JavaScript global types and methods, such as `Object`, `Function`, `eval`, etc. These APIs are portable across all standard ECMAScript environments, not just the Web, such as Node.js.

`web-sys` | [crates.io](#) | [repository](#)

Raw `wasm-bindgen` imports for all the Web's APIs, such as DOM manipulation, `setTimeout`, Web GL, Web Audio, etc.

Error Reporting and Logging

`console_error_panic_hook` | [crates.io](#) | [repository](#)

This crate lets you debug panics on `wasm32-unknown-unknown` by providing a panic hook that forwards panic messages to `console.error`.

`console_log` | [crates.io](#) | [repository](#)

This crate provides a backend for the `log` crate that routes logged messages to the devtools console.

Dynamic Allocation

`wee_alloc` | [crates.io](#) | [repository](#)

The **W**asm-**E**nabled, **E**lfin Allocator. A small (~1K uncompressed `.wasm`) allocator implementation for when code size is a greater concern than allocation performance.

Parsing and Generating `.wasm` Binaries

`parity-wasm` | [crates.io](#) | [repository](#)

Low-level WebAssembly format library for serializing, deserializing, and building `.wasm` binaries. Good support for well-known custom sections, such as the "names" section and "reloc.WHATEVER" sections.

`wasmparser` | [crates.io](#) | [repository](#)

A simple, event-driven library for parsing WebAssembly binary files. Provides the byte offsets of each parsed thing, which is necessary when interpreting relocs, for example.

Interpreting and Compiling WebAssembly

`wasmi` | [crates.io](#) | [repository](#)

An embeddable WebAssembly interpreter from Parity.

`cranelift-wasm` | [crates.io](#) | [repository](#)

Compile WebAssembly to the native host's machine code. Part of the Cranelift (né Cretonne) code generator project.

Tools You Should Know

This is a curated list of awesome tools you should know about when doing Rust and WebAssembly development.

Development, Build, and Workflow Orchestration

`wasm-pack` | [repository](#)

`wasm-pack` seeks to be a one-stop shop for building and working with Rust-generated WebAssembly that you would like to interoperate with JavaScript, on the Web or with Node.js.

`wasm-pack` helps you build and publish Rust-generated WebAssembly to the npm registry to be used alongside any other JavaScript package in workflows that you already use.

Optimizing and Manipulating `.wasm` Binaries

`wasm-opt` | [repository](#)

The `wasm-opt` tool reads WebAssembly as input, runs transformation, optimization, and/or instrumentation passes on it, and then emits the transformed WebAssembly as output.

Running it on the `.wasm` binaries produced by LLVM by way of `rustc` will usually create `.wasm` binaries that are both smaller and execute faster. This tool is a part of the `binaryen` project.

`wasm2js` | [repository](#)

The `wasm2js` tool compiles WebAssembly into "almost asm.js". This is great for supporting browsers that don't have a WebAssembly implementation, such as Internet Explorer 11. This tool is a part of the `binaryen` project.

`wasm-gc` | repository

A small tool to garbage collect a WebAssembly module and remove all unneeded exports, imports, functions, etc. This is effectively a `--gc-sections` linker flag for WebAssembly.

You don't usually need to use this tool yourself because of two reasons:

1. `rustc` now has a new enough version of `lld` that it supports the `--gc-sections` flag for WebAssembly. This is automatically enabled for LTO builds.
2. The `wasm-bindgen` CLI tool runs `wasm-gc` for you automatically.

`wasm-snip` | repository

`wasm-snip` replaces a WebAssembly function's body with an `unreachable` instruction.

Maybe you know that some function will never be called at runtime, but the compiler can't prove that at compile time? Snip it! Then run `wasm-gc` again and all the functions it transitively called (which could also never be called at runtime) will get removed too.

This is useful for forcibly removing Rust's panicking infrastructure in non-debug production builds.

Inspecting `.wasm` Binaries

`twiggy` | repository

`twiggy` is a code size profiler for `.wasm` binaries. It analyzes a binary's call graph to answer questions like:

- Why was this function included in the binary in the first place? I.e. which exported functions are transitively calling it?
- What is the retained size of this function? I.e. how much space would be saved if I removed it and all the functions that become dead code after its removal.

Use `twiggy` to make your binaries slim!

`wasm-objdump` | repository

Print low-level details about a `.wasm` binary and each of its sections. Also supports disassembling into the WAT text format. It's like `objdump` but for WebAssembly. This is a part of

the WABT project.

`wasm-nm` | repository

List the imported, exported, and private function symbols defined within a `.wasm` binary. It's like `nm` but for WebAssembly.

Project Templates

The Rust and WebAssembly working group curates and maintains a variety of project templates to help you kickstart new projects and hit the ground running.

`wasm-pack-template`

This template is for starting a Rust and WebAssembly project to be used with `wasm-pack`.

Use `cargo generate` to clone this project template:

```
cargo install cargo-generate
cargo generate --git https://github.com/rustwasm/wasm-pack-template.git
```

`create-wasm-app`

This template is for JavaScript projects that consume packages from npm that were created from Rust with `wasm-pack`.

Use it with `npm init`:

```
mkdir my-project
cd my-project/
npm init wasm-app
```

This template is often used alongside `wasm-pack-template`, where `wasm-pack-template` projects are installed locally with `npm link`, and pulled in as a dependency for a `create-wasm-app` project.

rust-webpack-template

This [template](#) comes pre-configured with all the boilerplate for compiling Rust to WebAssembly and hooking that directly into a Webpack build pipeline with Webpack's `rust-loader`.

Use it with `npm init`:

```
mkdir my-project
cd my-project/
npm init rust-webpack
```

Debugging Rust-Generated WebAssembly

This section contains tips for debugging Rust-generated WebAssembly.

Building with Debug Symbols

⚡ When debugging, always make sure you are building with debug symbols!

If you don't have debug symbols enabled, then the `"name"` custom section won't be present in the compiled `.wasm` binary, and stack traces will have function names like `wasm-function[42]` rather than the Rust name of the function, like

```
wasm_game_of_life::Universe::live_neighbor_count .
```

When using a "debug" build (aka `wasm-pack build --debug` or `cargo build`) debug symbols are enabled by default.

With a "release" build, debug symbols are not enabled by default. To enable debug symbols, ensure that you `debug = true` in the `[profile.release]` section of your `Cargo.toml`:

```
[profile.release]
debug = true
```

Logging with the `console` APIs

Logging is one of the most effective tools we have for proving and disproving hypotheses about why our programs are buggy. On the Web, the `console.log` function is the way to log messages to the browser's developer tools console.

We can use [the `web-sys` crate](#) to get access to the `console` logging functions:

```
extern crate web_sys;

web_sys::console::log_1(&"Hello, world!".into());
```

Alternatively, [the `console.error` function](#) has the same signature as `console.log`, but developer tools tend to also capture and display a stack trace alongside the logged message when `console.error` is used.

References

- Using `console.log` with the `web-sys` crate:
 - `web_sys::console::log` takes an array of values to log
 - `web_sys::console::log_1` logs a single value
 - `web_sys::console::log_2` logs two values
 - Etc...
- Using `console.error` with the `web-sys` crate:
 - `web_sys::console::error` takes an array of values to log
 - `web_sys::console::error_1` logs a single value
 - `web_sys::console::error_2` logs two values
 - Etc...
- The `console` object on MDN
- Firefox Developer Tools — Web Console
- Microsoft Edge Developer Tools — Console
- Get Started with the Chrome DevTools Console

Logging Panics

The `console_error_panic_hook` crate logs unexpected panics to the developer console via `console.error`. Rather than getting cryptic, difficult-to-debug `RuntimeError: unreachable executed` error messages, this gives you Rust's formatted panic message.

All you need to do is install the hook by calling `console_error_panic_hook::set_once()` in an initialization function or common code path:

```
#[wasm_bindgen]
pub fn init_panic_hook() {
    console_error_panic_hook::set_once();
}
```

Using a Debugger

Unfortunately, the debugging story for WebAssembly is still immature. On most Unix systems, [DWARF](#) is used to encode the information that a debugger needs to provide source-level inspection of a running program. There is an alternative format that encodes similar information on Windows. Currently, there is no equivalent for WebAssembly. Therefore, debuggers currently provide limited utility, and we end up stepping through raw WebAssembly instructions emitted by the compiler, rather than the Rust source text we authored.

There is a [sub-charter of the W3C WebAssembly group for debugging](#), so expect this story to improve in the future!

Nonetheless, debuggers are still useful for inspecting the JavaScript that interacts with our WebAssembly, and inspecting raw wasm state.

References

- [Firefox Developer Tools — Debugger](#)
- [Microsoft Edge Developer Tools — Debugger](#)
- [Get Started with Debugging JavaScript in Chrome DevTools](#)

Avoid the Need to Debug WebAssembly in the First Place

If the bug is specific to interactions with JavaScript or Web APIs, then [write tests with](#) `wasm-bindgen-test`.

If a bug does *not* involve interaction with JavaScript or Web APIs, then try to reproduce it as a normal Rust `#[test]` function, where you can leverage your OS's mature native tooling when debugging. Use testing crates like `quickcheck` and its test case shrinkers to mechanically reduce test cases. Ultimately, you will have an easier time finding and fixing bugs if you can isolate them in a smaller test cases that don't require interacting with JavaScript.

Note that in order to run native `#[test]` s without compiler and linker errors, you will need to ensure that `"rlib"` is included in the `[lib.crate-type]` array in your `Cargo.toml` file.

```
[lib]
crate-type ["cdylib", "rlib"]
```

Time Profiling

This section describes how to profile Web pages using Rust and WebAssembly where the goal is improving throughput or latency.

⚡ Always make sure you are using an optimized build when profiling! `wasm-pack build` will build with optimizations by default.

Available Tools

The `window.performance.now()` Timer

The `performance.now()` function returns a monotonic timestamp measured in milliseconds since the Web page was loaded.

Calling `performance.now` has little overhead, so we can create simple, granular measurements from it without distorting the performance of the rest of the system and inflicting bias upon our measurements.

We can use it to time various operations, and we can access `window.performance.now()` via the `web-sys` crate:

```
extern crate web_sys;

fn now() -> f64 {
    web_sys::window()
        .expect("should have a Window")
        .performance()
        .expect("should have a Performance")
        .now()
}
```

- The `web_sys::window` function
- The `web_sys::Window::performance` method
- The `web_sys::Performance::now` method

Developer Tools Profilers

All Web browsers' built-in developer tools include a profiler. These profilers display which functions are taking the most time with the usual kinds of visualizations like call trees and flame graphs.

If you [build with debug symbols](#) so that the "name" custom section is included in the wasm binary, then these profilers should display the Rust function names instead of something opaque like `wasm-function[123]`.

Note that these profilers *won't* show inlined functions, and since Rust and LLVM rely on inlining so heavily, the results might still end up a bit perplexing.

 [Screenshot of profiler with Rust symbols](#)

Resources

- [Firefox Developer Tools — Performance](#)
- [Microsoft Edge Developer Tools — Performance](#)
- [Chrome DevTools JavaScript Profiler](#)

The `console.time` and `console.timeEnd` Functions

The `console.time` and `console.timeEnd` functions allow you to log the timing of named operations to the browser's developer tools console. You call `console.time("some operation")` when the operation begins, and call `console.timeEnd("some operation")` when it finishes. The string label naming the operation is optional.

You can use these functions directly via the `web-sys` crate:

- `web_sys::console::time_with_label("some operation")`
- `web_sys::console::time_end_with_label("some operation")`

Here is a screenshot of `console.time` logs in the browser's console:

 [Screenshot of console.time logs](#)

Additionally, `console.time` and `console.timeEnd` logs will show up in your browser's profiler's "timeline" or "waterfall" view:

 [Screenshot of console.time logs](#)

Using `#[bench]` with Native Code

The same way we can often leverage our operating system's native code debugging tools by writing `#[test]` s rather than debugging on the Web, we can leverage our operating system's native code profiling tools by writing `#[bench]` functions.

Write your benchmarks in the `benches` subdirectory of your crate. Make sure that your `crate-type` includes `"rlib"` or else the bench binaries won't be able to link your main lib.

However! Make sure that you know the bottleneck is in the WebAssembly before investing much energy in native code profiling! Use your browser's profiler to confirm this, or else you risk wasting your time optimizing code that isn't hot.

Resources

- Using the `perf` profiler on Linux
- Using the Instruments.app profiler on macOS
- The VTune profiler supports Windows and Linux

Shrinking `.wasm` Code Size

This section will teach you how to optimize your `.wasm` build for a small code size footprint, and how to identify opportunities to change your Rust source such that less `.wasm` code is emitted.

Why Care About Code Size?

When serving a `.wasm` file over the network, the smaller it is, the faster the client can download it. Faster `.wasm` downloads lead to faster page load times, and that leads to happier users.

However, it's important to remember though that code size likely isn't the end-all-be-all metric you're interested in, but rather something much more vague and hard to measure like "time to first interaction". While code size plays a large factor in this measurement (can't do anything if you don't even have all the code yet!) it's not the only factor.

WebAssembly is typically served to users gzip'd so you'll want to be sure to compare differences in gzip'd size for transfer times over the wire. Also keep in mind that the WebAssembly binary format is quite amenable to gzip compression, often getting over 50% reductions in size.

Furthermore, WebAssembly's binary format is optimized for very fast parsing and processing. Browsers nowadays have "baseline compilers" which parses WebAssembly and emits compiled code as fast as wasm can come in over the network. This means that [if you're using](#)

`instantiateStreaming` the second the Web request is done the WebAssembly module is probably ready to go. JavaScript, on the other hand, can often take longer to not only parse but also get up to speed with JIT compilation and such.

And finally, remember that WebAssembly is also far more optimized than JavaScript for execution speed. You'll want to be sure to measure for runtime comparisons between JavaScript and WebAssembly to factor that in to how important code size is.

All this to say basically don't dismay immediately if your `.wasm` file is larger than expected! Code size may end up only being one of many factors in the end-to-end story. Comparisons between JavaScript and WebAssembly that only look at code size are missing the forest for the trees.

Optimizing Builds for Code Size

There are a bunch of configuration options we can use to get `rustc` to make smaller `.wasm` binaries. In some cases, we are trading longer compile times for smaller `.wasm` sizes. In other cases, we are trading runtime speed of the WebAssembly for smaller code size. We should be cognizant of the trade offs of each option, and in the cases where we trade runtime speed for code size, profile and measure to make an informed decision about whether the trade is worth it.

Compiling with Link Time Optimizations (LTO)

In `Cargo.toml`, add `lto = true` in the `[profile.release]` section:

```
[profile.release]
lto = true
```

This gives LLVM many more opportunities to inline and prune functions. Not only will it make the `.wasm` smaller, but it will also make it faster at runtime! The downside is that compilation will take longer.

Tell LLVM to Optimize for Size Instead of Speed

LLVM's optimization passes are tuned to improve speed, not size, by default. We can change the goal to code size by modifying the `[profile.release]` section in `Cargo.toml` to this:

```
[profile.release]
opt-level = 's'
```


Or, to even more aggressively optimize for size, at further potential speed costs:

```
[profile.release]
opt-level = 'z'
```

Note that, surprisingly enough, `opt-level = "s"` can sometimes result in smaller binaries than `opt-level = "z"`. Always measure!

Use the `wasm-opt` Tool

The [Binaryen](#) toolkit is a collection of WebAssembly-specific compiler tools. It goes much further than LLVM's WebAssembly backend does, and using its `wasm-opt` tool to post-process a `.wasm` binary generated by LLVM can often get another 15-20% savings on code size. It will often produce runtime speed ups at the same time!

```
# Optimize for size.
wasm-opt -Os -o output.wasm input.wasm

# Optimize aggressively for size.
wasm-opt -Oz -o output.wasm input.wasm

# Optimize for speed.
wasm-opt -O -o output.wasm input.wasm

# Optimize aggressively for speed.
wasm-opt -O3 -o output.wasm input.wasm
```

Notes about Debug Information

One of the biggest contributors to wasm binary size can be debug information and the `names` section of the wasm binary. The `wasm-pack` tool, however, removes debuginfo by default. Additionally `wasm-opt` removes the `names` section by default unless `-g` is also specified.

This means that if you follow the above steps you should by default not have either debuginfo or the names section in the wasm binary. If, however, you are manually otherwise preserving this debug information in the wasm binary be sure to be mindful of this!

Size Profiling

If tweaking build configurations to optimize for code size isn't resulting in a small enough `.wasm` binary, it is time to do some profiling to see where the remaining code size is coming from.

⚡ Just like how we let time profiling guide our speed up efforts, we want to let size profiling guide our code size shrinking efforts. Fail to do this and you risk wasting your own time!

The `twiggy` Code Size Profiler

`twiggy` is a code size profiler that supports WebAssembly as input. It analyzes a binary's call graph to answer questions like:

- Why was this function included in the binary in the first place?
- What is the *retained size* of this function? I.e. how much space would be saved if I removed it and all the functions that become dead code after its removal?

```
$ twiggy top -n 20 pkg/wasm_game_of_life_bg.wasm
```

Shallow Bytes	Shallow %	Item
9158	19.65%	"function names" subsection
3251	6.98%	dlmalloc::dlmalloc::Dlmalloc::malloc::h632d10c184fef6e8
2510	5.39%	<str as core::fmt::Debug>::fmt::he0d87479d1c208ea
1737	3.73%	data[0]
1574	3.38%	data[3]
1524	3.27%	core::fmt::Formatter::pad::h6825605b326ea2c5
1413	3.03%	std::panicking::rust_panic_with_hook::h1d3660f2e339513d
1200	2.57%	core::fmt::Formatter::pad_integral::h06996c5859a57ced
1131	2.43%	core::str::slice_error_fail::h6da90c14857ae01b
1051	2.26%	core::fmt::write::h03ff8c7a2f3a9605
931	2.00%	data[4]
864	1.85%	dlmalloc::dlmalloc::Dlmalloc::free::h27b781e3b06bdb05
841	1.80%	<char as core::fmt::Debug>::fmt::h07742d9f4a8c56f2
813	1.74%	__rust_realloc
708	1.52%	core::slice::memchr::memchr::h6243a1b2885fdb85
678	1.45%	<core::fmt::builders::PadAdapter<'a> as core::fmt::Write>::write_str::h96b72fb7457d3062
631	1.35%	universe_tick
631	1.35%	
		dlmalloc::dlmalloc::Dlmalloc::dispose_chunk::hae6c5c8634e575b8
514	1.10%	std::panicking::default_hook::{{closure}}::hfae0c204085471d5
503	1.08%	<&'a T as core::fmt::Debug>::fmt::hba207e4f7abaece6

Manually Inspecting LLVM-IR

LLVM-IR is the final intermediate representation in the compiler toolchain before LLVM generates WebAssembly. Therefore, it is very similar to the WebAssembly that is ultimately emitted. More LLVM-IR generally means more `.wasm` size, and if a function takes up 25% of the

LLVM-IR, then it generally will take up 25% of the `.wasm`. While these numbers only hold in general, the LLVM-IR has crucial information that is not present in the `.wasm` (because of WebAssembly's lack of a debugging format like DWARF): which subroutines were inlined into a given function.

You can generate LLVM-IR with this `cargo` command:

```
cargo rustc --release -- --emit llvm-ir
```

Then, you can use `find` to locate the `.ll` file containing the LLVM-IR in `cargo`'s `target` directory:

```
find target/release -type f -name '*.ll'
```

References

- [LLVM Language Reference Manual](#)

More Invasive Tools and Techniques

Tweaking build configurations to get smaller `.wasm` binaries is pretty hands off. When you need to go the extra mile, however, you are prepared to use more invasive techniques, like rewriting source code to avoid bloat. What follows is a collection of get-your-hands-dirty techniques you can apply to get smaller code sizes.

Avoid String Formatting

`format!`, `to_string`, etc... can bring in a lot of code bloat. If possible, only do string formatting in debug mode, and in release mode use static strings.

Avoid Panicking

This is definitely easier said than done, but tools like `twiggy` and manually inspecting LLVM-IR can help you figure out which functions are panicking.

Panics do not always appear as a `panic!()` macro invocation. They arise implicitly from many constructs, such as:

- Indexing a slice panics on out of bounds indices: `my_slice[i]`

- Division will panic if the divisor is zero: `dividend / divisor`
- Unwrapping an `Option` or `Result`: `opt.unwrap()` or `res.unwrap()`

The first two can be translated into the third. Indexing can be replaced with fallible `my_slice.get(i)` operations. Division can be replaced with `checked_div` calls. Now we only have a single case to contend with.

Unwrapping an `Option` or `Result` without panicking comes in two flavors: safe and unsafe.

The safe approach is to `abort` instead of panicking when encountering a `None` or an `Error`:

```
#[inline]
pub fn unwrap_abort<T>(o: Option<T>) -> T {
    use std::process;
    match o {
        Some(t) => t,
        None => process::abort(),
    }
}
```

Ultimately, panics translate into aborts in `wasm32-unknown-unknown` anyways, so this gives you the same behavior but without the code bloat.

Alternatively, the `unreachable` crate provides an unsafe `unchecked_unwrap` extension method for `Option` and `Result` which tells the Rust compiler to *assume* that the `Option` is `Some` or the `Result` is `Ok`. It is undefined behavior what happens if that assumption does not hold. You really only want to use this unsafe approach when you 110% *know* that the assumption holds, and the compiler just isn't smart enough to see it. Even if you go down this route, you should have a debug build configuration that still does the checking, and only use unchecked operations in release builds.

Avoid Allocation or Switch to `wee_alloc`

Rust's default allocator for WebAssembly is a port of `dlmalloc` to Rust. It weighs in somewhere around ten kilobytes. If you can completely avoid dynamic allocation, then you should be able to shed those ten kilobytes.

Completely avoiding dynamic allocation can be very difficult. But removing allocation from hot code paths is usually much easier (and usually helps make those hot code paths faster, as well). In these cases, [replacing the default global allocator with `wee_alloc`](#) should save you most (but not quite all) of those ten kilobytes. `wee_alloc` is an allocator designed for situations where you need *some* kind of allocator, but do not need a particularly fast allocator, and will happily trade allocation speed for smaller code size.

Use Trait Objects Instead of Generic Type Parameters

When you create generic functions that use type parameters, like this:

```
fn whatever<T: MyTrait>(t: T) { ... }
```

Then `rustc` and LLVM will create a new copy of the function for each `T` type that the function is used with. This presents many opportunities for compiler optimizations based on which particular `T` each copy is working with, but these copies add up quickly in terms of code size.

If you use trait objects instead of type parameters, like this:

```
fn whatever(t: Box<MyTrait>) { ... }  
// or  
fn whatever(t: &MyTrait) { ... }  
// etc...
```

Then dynamic dispatch via virtual calls is used, and only a single version of the function is emitted in the `.wasm`. The downside is the loss of the compiler optimization opportunities and the added cost of indirect, dynamically dispatched function calls.

Use the `wasm-snip` Tool

`wasm-snip` replaces a WebAssembly function's body with an `unreachable` instruction. This is a rather heavy, blunt hammer for functions that kind of look like nails if you squint hard enough.

Maybe you know that some function will never be called at runtime, but the compiler can't prove that at compile time? Snip it! Afterwards, run `wasm-opt` again with the `--dce` flag, and all the functions that the snipped function transitively called (which could also never be called at runtime) will get removed too.

This tool is particularly useful for removing the panicking infrastructure, since panics ultimately translate into traps anyways.

JavaScript Interoperation

Importing and Exporting JS Functions

From the Rust Side

When using wasm within a JS host, importing and exporting functions from the Rust side is straightforward: it works very similarly to C.

WebAssembly modules declare a sequence of imports, each with a *module name* and an *import name*. The module name for an `extern { ... }` block can be specified using

`#[link(wasm_import_module)]`, currently it defaults to "env".

Exports have only a single name. In addition to any `extern` functions the WebAssembly instance's default linear memory is exported as "memory".

```
// import a JS function called `foo` from the module `mod`  
#[link(wasm_import_module = "mod")]  
extern { fn foo(); }  
  
// export a Rust function called `bar`  
#[no_mangle]  
pub extern fn bar() { /* ... */ }
```

Because of wasm's limited value types, these functions must operate only on primitive numeric types.

From the JS Side

Within JS, a wasm binary turns into an ES6 module. It must be *instantiated* with linear memory and have a set of JS functions matching the expected imports. The details of instantiation are available on [MDN](#).

The resulting ES6 module will contain all of the functions exported from Rust, now available as JS functions.

[Here](#) is a very simple example of the whole setup in action.

Going Beyond Numerics

When using wasm within JS, there is a sharp split between the wasm module's memory and the JS memory:

- Each wasm module has a linear memory (described at the top of this document), which is initialized during instantiation. **JS code can freely read and write to this memory.**
- By contrast, wasm code has no *direct* access to JS objects.

Thus, sophisticated interop happens in two main ways:

- Copying in or out binary data to the wasm memory. For example, this is one way to provide an owned `String` to the Rust side.
- Setting up an explicit "heap" of JS objects which are then given "addresses". This allows wasm code to refer to JS objects indirectly (using integers), and operate on those objects by invoking imported JS functions.

Fortunately, this interop story is very amenable to treatment through a generic "bindgen"-style framework: [wasm-bindgen](#). The framework makes it possible to write idiomatic Rust function signatures that map to idiomatic JS functions, automatically.

Custom Sections

Custom sections allow embedding named arbitrary data into a wasm module. The section data is set at compile time and is read directly from the wasm module, it cannot be modified at runtime.

In Rust, custom sections are static arrays (`[T; size]`) exposed with the `#[link_section]` attribute:

```
#[link_section = "hello"]
pub static SECTION: [u8; 24] = *b"This is a custom section";
```

This adds a custom section named `hello` to the wasm file, the rust variable name `SECTION` is arbitrary, changing it wouldn't alter the behaviour. The contents are bytes of text here but could be any arbitrary data.

The custom sections can be read on the JS side using the `WebAssembly.Module.customSections` function, it takes a wasm Module and the section name as arguments and returns an Array of `ArrayBuffer`s. Multiple sections may be specified using the same name, in which case they will all appear in this array.

```
WebAssembly.compileStreaming(fetch("sections.wasm"))
  .then(mod => {
    const sections = WebAssembly.Module.customSections(mod, "hello");

    const decoder = new TextDecoder();
    const text = decoder.decode(sections[0]);

    console.log(text); // -> "This is a custom section"
  });
```

Which Crates Will Work Off-the-Shelf with WebAssembly?

It is easiest to list the things that do *not* currently work with WebAssembly; crates which avoid these things tend to be portable to WebAssembly and usually *Just Work*. A good rule of thumb is that if a crate supports embedded and `#![no_std]` usage, it probably also supports WebAssembly.

Things a Crate Might do that Won't Work with WebAssembly

C and System Library Dependencies

There are no system libraries in wasm, so any crate that tries to bind to a system library won't work.

Using C libraries will also probably fail to work, since wasm doesn't have a stable ABI for cross-language communication, and cross-language linking for wasm is very finicky. Everyone wants this to work eventually, especially since `clang` is shipping their `wasm32` target by default now, but the story isn't quite there yet.

File I/O

WebAssembly does not have access to a file system, so crates that assume the existence of a file system — and don't have wasm-specific workarounds — will not work.

Spawning Threads

There are [plans to add threading to WebAssembly](#), but it isn't shipping yet. Attempts to spawn on a thread on the `wasm32-unknown-unknown` target will panic, which triggers a wasm trap.

So Which General Purpose Crates Tend to Work Off-the-Shelf with WebAssembly?

Algorithms and Data Structures

Crates that provide the implementation of a particular [algorithm](#) or [data structure](#), for example A* graph search or splay trees, tend to work well with WebAssembly.

```
#![no_std]
```

Crates that do not rely on the standard library tend to work well with WebAssembly.

Parsers

[Parsers](#) — so long as they just take input and don't perform their own I/O — tend to work well with WebAssembly.

Text Processing

Crates that deal with the complexities of human language when expressed in textual form tend to work well with WebAssembly.

Rust Patterns

Shared solutions for particular situations specific to programming in Rust tend to work well with WebAssembly.

How to Add WebAssembly Support to a General-Purpose Crate

This section is for general-purpose crate authors who want to support WebAssembly.

Maybe Your Crate Already Supports WebAssembly!

Review the information about [what kinds of things can make a general-purpose crate *not* portable for WebAssembly](#). If your crate doesn't have any of those things, it likely already supports WebAssembly!

You can always check by running `cargo build` for the WebAssembly target:

```
cargo build --target wasm32-unknown-unknown
```

If that command fails, then your crate doesn't support WebAssembly right now. If it doesn't fail, then your crate *might* support WebAssembly. You can be 100% sure that it does (and continues to do so!) by [adding tests for wasm and running those tests in CI](#).

Adding Support for WebAssembly

Avoid Performing I/O Directly

On the Web, I/O is always asynchronous, and there isn't a file system. Factor I/O out of your library, let users perform the I/O and then pass the input slices to your library instead.

For example, refactor this:

```
use std::fs;
use std::path::Path;

pub fn parse_thing(path: &Path) -> Result<MyThing, MyError> {
    let contents = fs::read(path)?;
    // ...
}
```

Into this:

```
pub fn parse_thing(contents: &[u8]) -> Result<MyThing, MyError> {
    // ...
}
```

Add `wasm-bindgen` as a Dependency

If you need to interact with the outside world (i.e. you can't have library consumers drive that interaction for you) then you'll need to add `wasm-bindgen` (and `js-sys` and `web-sys` if you need them) as a dependency for when compilation is targeting WebAssembly:

```
[target.'cfg(target_arch = "wasm32")'.dependencies]
wasm-bindgen = "0.2"
js-sys = "0.3"
web-sys = "0.3"
```

Avoid Synchronous I/O

If you must perform I/O in your library, then it cannot be synchronous. There is only asynchronous I/O on the Web. Use [the `futures` crate](#) and [the `wasm-bindgen-futures` crate](#) to manage asynchronous I/O. If your library functions are generic over some future type `F`, then that future can be implemented via `fetch` on the Web or via non-blocking I/O provided by the operating system.

```
pub fn do_stuff<F>(future: F) -> impl Future<Item = MyOtherThing>
where
    F: Future<Item = MyThing>,
{
    // ...
}
```

You can also define a trait and implement it for WebAssembly and the Web and also for native targets:

```
trait ReadMyThing {
    type F: Future<Item = MyThing>;
    fn read(&self) -> Self::F;
}

#[cfg(target_arch = "wasm32")]
struct WebReadMyThing {
    // ...
}

#[cfg(target_arch = "wasm32")]
impl ReadMyThing for WebReadMyThing {
    // ...
}

#[cfg(not(target_arch = "wasm32"))]
struct NativeReadMyThing {
    // ...
}

#[cfg(not(target_arch = "wasm32"))]
impl ReadMyThing for NativeReadMyThing {
    // ...
}
```

Avoid Spawning Threads

Wasm doesn't support threads yet (but [experimental work is ongoing](#)), so attempts to spawn threads in wasm will panic.

You can use `#[cfg(...)]`s to enable threaded and non-threaded code paths depending on if the target is WebAssembly or not:

```
#[cfg(target_arch = "wasm32")]
fn do_work() {
    // Do work with only this thread...
}

#[cfg(not(target_arch = "wasm32"))]
fn do_work() {
    use std::thread;

    // Spread work to helper threads....
    thread::spawn(|| {
        // ...
    });
}
```

Another option is to factor out thread spawning from your library and allow users to "bring their own threads" similar to factoring out file I/O and allowing users to bring their own I/O. This has the side effect of playing nice with applications that want to own their own custom thread pool.

Maintaining Ongoing Support for WebAssembly

Building for `wasm32-unknown-unknown` in CI

Ensure that compilation doesn't fail when targeting WebAssembly by having your CI script run these commands:

```
rustup target add wasm32-unknown-unknown
cargo check --target wasm32-unknown-unknown
```

For example, you can add this to your `.travis.yml` configuration for Travis CI:


```
matrix:
  include:
    - language: rust
      rust: stable
      name: "check wasm32 support"
      install: rustup target add wasm32-unknown-unknown
      script: cargo check --target wasm32-unknown-unknown
```

Testing in Node.js and Headless Browsers

You can use `wasm-bindgen-test` and the `wasm-pack test` subcommand to run wasm tests in either Node.js or a headless browser. You can even integrate these tests into your CI.

[Learn more about testing wasm here.](#)

Deploying Rust and WebAssembly to Production

 **Deploying Web applications built with Rust and WebAssembly is nearly identical to deploying any other Web application!**

To deploy a Web application that uses Rust-generated WebAssembly on the client, copy the built Web application's files to your production server's file system and configure your HTTP server to make them accessible.

Ensure that Your HTTP Server Uses the `application/wasm` MIME Type

For the fastest page loads, you'll want to use the `WebAssembly.instantiateStreaming` function to pipeline wasm compilation and instantiation with network transfer (or make sure your bundler is able to use that function). However, `instantiateStreaming` requires that the HTTP response has the `application/wasm` MIME type set, or else it will throw an error.

- [How to configure MIME types for the Apache HTTP server](#)
- [How to configure MIME types for the NGINX HTTP server](#)

More Resources

- [Best Practices for Webpack in Production](#). Many Rust and WebAssembly projects use Webpack to bundle their Rust-generated WebAssembly, JavaScript, CSS, and HTML. This guide has tips for getting the most out of Webpack when deploying to production environments.
- [Apache documentation](#). Apache is a popular HTTP server for use in production.
- [NGINX documentation](#). NGINX is a popular HTTP server for use in production.