Check old ver

Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, September 24, 2022. Git commit: 42748ab

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
```

```
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↙
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 1

High-performance web frontend using WebAssembly

We've seen how Rust can help us in the backend in many different ways: static web servers, REST APIs, serverless computing, WebSocket. But can you use Rust in the frontend? The answer is yes! With the introduction of *WebAssembly*[1] (abbreviated *Wasm*),

[1] https://webassembly.org/

you can compile a Rust program to WebAssembly, and run it in browsers, alongside JavaScript.

1.1 What is WebAssembly

WebAssembly is an open standard for a binary instruction format that runs on a stack-based virtual machine. Its original design goal was to provide near-native performance in web browsers. You can think of it as an assembly language for the web. WebAssembly is a World Wide Web Consortium (W3C) recommendation, and it's implemented in all major browsers.

WebAssembly is designed to run at near-native speed. It doesn't require you to use a garbage collector (GC)[2]. It can be a compile target for many languages, like C, C++, and Rust. Therefore, you can write frontend applications in the high-level programming language you prefer, and get predictable performance.

There are a few reasons why you might want to use Rust to compile to WebAssembly:

[2] Although there are discussions underway to add GC as an optional feature.

- Enjoy the high-level syntax and low-level control of Rust in browsers
- Save bandwidth while downloading the small `.wasm` binary because of Rust's minimal runtime
- Reuse the extensive collection of existing Rust libraries
- Use familiar frontend tools, like ES6 modules, npm, webpack, through the `wasm-pack` toolchain

There are also some common misconceptions about WebAssembly:

- WebAssembly is not aiming to replace JavaScript completely. It is supposed to run alongside JavaScript and complement each other.
- WebAssembly is not only limited to the browsers, although it was initially targeting the browser. The WebAssembly runtime can potentially run anywhere. For example, on the server-side or in IoT devices.

A common use case for WebAssembly is to speed up the performance bottleneck of JavaScript web applications. The user interface (UI) can be built in HTML, CSS, and JavaScript. But when the application needs to execute CPU-intensive jobs, it calls WebAssembly. The result of the computation can then be passed back to JavaScript for display.

Some framework takes this idea further to let you write the whole frontend application in Rust. They usually take inspiration from other popular frontend frameworks like React and Elm and use a Virtual DOM[3]. The Rust code is compiled to Wasm and rendered to the screen by the Virtual DOM.

1.2 What are you building?

First, you'll be building a Hello World application. This application will create a browser `alert()` from Rust. This example will show you the process of getting a WebAssembly program up and running. You'll also learn how WebAssembly works with JavaScript.

[3] https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom

In the Catdex example from Chapter **??**, you allow users to upload cat photos. But the user might upload a very high-resolution photo that takes a lot of bandwidth. To save bandwidth, you can resize the photo in the frontend before upload. But image resizing is a CPU-intensive job, so it makes sense to implement the resize algorithm in WebAssembly. You'll be building a frontend application for reducing the size of a cat image.

Once you get hold of how WebAssembly can work with JavaScript, you can start to use a fully Rust frontend framework. You'll first start with a hello-world-style example to get familiar with the setup and build process. This example will have a button that can increase a counter.

Finally, you'll be rebuilding the cat photo resize application with the Yew[4] framework.

1.3 Hello WebAssembly!

There are quite a few steps to run a Hello World program in WebAssembly. Conceptually, this is how you get some Rust code running in the browser as WebAssembly:

[4] https://yew.rs

1. Write the Rust code to expose functionality to JavaScript, and to handle data passing between JavaScript and wasm.

2. Use the compiler toolchain to compile Rust code into a `.wasm` binary

3. Serve the `.wasm` file on a web server

4. Write an HTML and JavaScript page to load this `.wasm` file.

5. In the JavaScript file, `fetch`[5] the `.wasm` file and use the `WebAssembly.instantiateStreaming()`[6] API to compile and instantiate the `.wasm` module.

6. In JavaScript, make calls to the functions which the `.wasm` module exports.

These steps are tedious and do not feel as ergonomic as what `cargo` or `npm` offer. Thankfully there is a tool called `wasm-pack` that bundles many tools that make this

[5] `Fetch` is a web API that allows you to download additional resources. It's a successor of the old `XMLHttpRequest`

[6] Check https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running for more detail.

process smoother. Also, to avoid writing boilerplate code, you can use the template `wasm-pack-template`[7] to quickly generate a project.

Setting up the development environment

To set up `wasm-pack`, head to `https://rustwasm.github.io/wasm-pack/installer/`. For Linux, it's as simple as executing the following command in the terminal[8]:

```
curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf
  | sh
```

TODO: mention npm version `Wasm-pack` helps you package the project into an npm (Node Package Manager) package, so developers who are familiar with modern JavaScript development can easily pick it up. To properly package and publish the package, you need to install the command-line `npm` the same way as in Chapter **??**.

[7] https://github.com/rustwasm/wasm-pack-template

[8] `curl` is a popular command-line HTTP client. If you don't have it yet you can almost certainly find it in your Linux distribution's package directory.

Finally, you need to install `cargo-generate`, a cargo subcommand that helps you create new projects using templates. Simply run this command in the command line:

```
cargo install cargo-generate
```

Creating the project

Now you have all the required tools installed. You can start creating the project by running:

```
wasm-pack new hello-wasm
```

TODO: Explain that wasm-pack new now uses cargo generate under the hood This command makes `cargo-generate` download the `wasm-pack-template` template for GitHub and create a project locally. `Cargo-generate` will ask you for the project name, you can name it "hello-wasm". After `cargo-generate` finishes, you'll see a `hello-wasm` folder in the current directory.

TODO: Explain this output

```
[INFO]: Installing cargo-generate...
 Generating a new rustwasm project with name 'hello-wasm'...
 Creating project called `hello-wasm`...
 Done! New project created /home/user/hello-wasm
[INFO]: Generated new project at /hello-wasm
```

In the `hello-wasm` folder, you'll find a fairly typical cargo library project, with a `Cargo.toml` and `src/lib.rs`. But if you look closely into the `Cargo.toml` file Listing **??**, you'll see it has a few interesting features[9]:

```
[package]
name = "hello-wasm"
version = "0.1.0"
authors = ["Shing Lyu"]
edition = "2018"
```

[9] The `wasm-pack-template` is being updated from time to time. The versions of the dependencies might be newer than the ones listed here.

```
[lib]

crate-type = ["cdylib", "rlib"]


[features]

default = ["console_error_panic_hook"]


[dependencies]

wasm-bindgen = "0.2.63"



# The `console_error_panic_hook` crate provides better debugging
    ↪ of panics by
# logging them with `console.error`. This is great for
    ↪ development, but requires
# all the `std::fmt` and `std::panicking` infrastructure, so isn
```

```
    ↪ 't great for
# code size when deploying.
console_error_panic_hook = { version = "0.1.6", optional = true
    ↪ }

# `wee_alloc` is a tiny allocator for wasm that is only ~1K in
    ↪ code size
# compared to the default allocator's ~10K. It is slower than
    ↪ the default
# allocator, however.
#
# Unfortunately, `wee_alloc` requires nightly Rust when
    ↪ targeting wasm for now.
wee_alloc = { version = "0.4.5", optional = true }
```

```
[dev-dependencies]
wasm-bindgen-test = "0.3.13"
```

check old.

```
[profile.release]
# Tell `rustc` to optimize for small code size.
opt-level = "s"
```

TODO: Explain manual update to version 2021

The `crate-type` is `cdylib` (**C** **Dy**namic **Lib**rary) and `rlib` (**R**ust **Lib**rary). `Cdylib` ensures that the output is a dynamic library that follows the C FFI convention. All the Rust-specific information isstripped away. This will help the LLVM compiler that compiles our code to Wasm understand the exported interfaces. `Rlib` is added to for running unit tests, it's not required for compiling to WebAssembly.

Since the browsers will download the `.wasm` binary through the internet, it's crucial to keep the binary size small, so the download is fast. You'll notice that in `[profile.`

release], the opt-level options is set to s, which means optimize for small code size. The template also chooses to use a custom memory allocator wee_alloc that is optimized for code size.

It also adds the wasm-bindgen crate, which is used to generate binding between WebAssembly and JavaScript. You can see the wasm-bindgen crate being used in the src/lib.rs file (Listing ??).

```
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, hello-wasm!");
}
```

The first few line in `src/lib.rs` sets up the `wee_alloc` allocator, with which you won't go into detail.

The next two blocks are the key in this hello world example. What this file is trying to do is:

1. Expose the JavaScript DOM API `window.alert()` to Rust/Wasm

2. Expose a Wasm function named `greet()` to JavaScript.

3. When JavaScript calls the `greet()` Wasm function, call the `alert()` function from Wasm to display a pop up message in the browser.

The following block in Listing **??** exposes the `window.alert()` function to Wasm:

```
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}
```

The `extern` block tells Rust this function defines as a foreign function interface (FFI). Rust can call this foreign JavaScript function defined elsewhere.

Notice that the `alert` function takes a `&str`. This matches the JavaScript `alert`, which takes a JS `String`. However, in Wasm's specification, you are only allowed to pass integers and floating-point numbers across JavaScript and Wasm. So how can you pass a `&str` as the parameter? This is the magic of `wasm_bindgen`.

The `#[wasm_bindgen]` attribute tells `wasm_bindgen` to create a binding. `Wasm_bindgen` generates Wasm code that encodes the `&str` into an integer array, passes it to JavaScript, then generates JavaScript code that converts the integer array back into a JavaScript string.

`Wasm_bindgen` works the other way around: you can expose a Rust function using `pub fn greet()` and annotate it with the `#[wasm_bindgen]` attribute. `Wasm_bindgen` will compile this function to Wasm and expose it to JavaScript.

■ **NOTE** You might be wondering what is the purpose of `src/utils.rs` and the `console_error_panic_hook` feature defined in `Cargo.toml`. When Rust code panics, you'll only see a generic Wasm error message in the browser's console. The `console_error_panic_hook` feature prints a more informative error message about the panic to the browser's console, which helps you with debugging. The `console_error_panic_hook` needs to be explicitly initialized once, and so the `src/utils.rs` provides a small function to do that.

If you now run `wasm-pack build`, `wasm-pack` will ensure that you have the correct toolchain (for example, download the correct compilation target with rustup) and compile your code to Wasm. You'll see the output in the `pkg` folder. `Wasm-pack` generates a few files:

- `hello_wasm_bg.wasm`: The compiled Wasm binary, containing the Rust function you exposed.
- `hello_wasm.js`: Some JavaScript binding wrapper around the Wasm functions that makes passing values easier.
- `hello_wasm_bg.d.ts`: TypeScript type definition. Useful if you want to develop the frontend in TypeScript.
- `hello_wasm.d.ts`: TypeScript definition.
- `package.json`: The npm project metadata file. This will be useful when you publish the package to npm.
- `README.md`: A short introductory note to the package user. It will be shown on

the npm website if you publish this package.

■ **NOTE** TypeScript is a programming language that builds on JavaScript by adding static type definitions. As a Rust developer, you already know the power of static types. Since the Rust code you write for Wasm is typed, it makes sense to use it with typed TypeScript instead of JavaScript so that you can enjoy the power of static typing end-to-end.

`Wasm-pack` doesn't force you to use TypeScript, so it generates a `.js` file containing the implementation, and a `.d.ts` definition file that contains TypeScript type definitions. If the frontend uses JavaScript, it can use the `.js` file only and ignore the `.d.ts` file. But if the frontend uses TypeScript it can reference the `.d.ts` file to enforce the types.

Because TypeScript is a topic that deserves its own book, I'll stick with JavaScript in this book.

Creating the frontend

Now you have the Wasm package ready, but how do you make it work on a web page? Since Wasm does not support ES6 `import` statement yet, you'll have to perform a `fetch` to download the `.wasm` file, then call the `WebAssembly.instantiateStreaming()` web API to instantiate it. This is quite cumbersome and doesn't feel natural to the `npm`-style workflow. Instead, you can use `Webpack` to simplify the way you import the Wasm package into a JavaScript application.

`Webpack` is a versatile tool for bundling your JavaScript files. It can analyze the dependency of your various JavaScript file and packages installed from npm, and package them into a single `.js` file. This reduces the overhead of downloading multiple JavaScript files, and reduce the risk of missing dependencies in runtime. The most important feature you want from `Webpack` is using ES6 `import` statement to import a Wasm package. This allows you to avoid all the boilerplate code of `fetch`ing the `.wasm` file and instantiating it.

`Webpack` requires some configuration to work with Wasm. To save you this trouble, you are going to use another template, `create-wasm-app`[10]. This template creates a frontend web page project with Webpack configuration for Wasm. To initiate a project based on this template, simply run the following command in the command-line inside the `hello-wasm` folder:

```
npm init wasm-app client
```

This command will download the `create-wasm-app` template[11] and create the project in a folder called `client`.

■ **TIP** When you run `cargo generate`, cargo will initialize a git project in the created project directory. When you run `npm init wasm-app client`, npm will also initialize a separate git repository inside the `client` folder. So you end up with two git repositories, one inside the other. If you want to version control the whole project in

[10] https://github.com/rustwasm/create-wasm-app

[11] A npm template, officially called an *initializer*, is a npm package with a prefix `create-` in the name. The command `npm init foo` is a shorthand for `npm init create-foo`. `npm` will look for the npm package named `create-foo`.

one git repository, you can delete the inner `client/.git` folder.

Since this template creates a frontend project, there should be an HTML file as the entry point. You can find an `index.html` file in the `client` folder, shown in Listing **??**.

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello wasm-pack!</title>
  </head>
  <body>
    <noscript>
      This page contains webassembly and javascript content,
      please enable javascript in your browser.
    </noscript>
    <script src="./bootstrap.js"></script>
  </body>
```

```
</html>
```

The `index.html` is a very minimal HTML page. It includes the `bootstrap.js` file Listing ?? with a `<script>` tag.

```
// A dependency graph that contains any wasm must
// all be imported asynchronously. This `bootstrap.js`
// file does the single async import, so that no one
// else needs to worry about it again.
import("./index.js")
  .catch(e => console.error("Error importing `index.js`:", e));
```

This `bootstrap.js` file imports the `index.js` file asynchronously. This is the limitation of Webpack v4, such that the file can not be imported synchronously. The `index.js` file Listing ?? is what actually uses the Wasm package.

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

```
wasm.greet();
```

In `index.js`, the template imports a demo Wasm package on npm called `hello-wasm-pack`. But you want to use the Wasm project you just built in the parent directly. How do you change that? You'll need to open the `package.json` file and add a `dependencies` section as shown in Listing **??**.

```
{
  "name": "create-wasm-app",
  // ...
  "dependencies": {
    "hello-wasm": "file:../pkg"
  },
  "devDependencies": {
    // Removed the hello-wasm-pack package
    "webpack": "^4.29.3",
    "webpack-cli": "^3.1.0",
```

```
    "webpack-dev-server": "^3.1.5",
    "copy-webpack-plugin": "^5.0.0"
  }
}
```

In `dependencies`, you defined a new package called `hello-wasm`, and the `file:../pkg` means the package is located in the same file system, in the `../pkg` folder. Don't forget to remove the unused `hello-wasm-pack` demo package from `devDependencies` as well.

Then you can go back to Listing **??** and change the first line to:

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

This will load the `hello-wasm` package. The next line calls the `greet` function you exported from Rust:

```
wasm.greet();
```

As mentioned before, the `import` statement won't work without Webpack. This template already has all the Webpack configuration you have, including:

- `webpack.config.js`: Webpack-specific configurations.
- `package.json`
  - `devDependencies`: this section specifies all the dependencies like `webpack`, `webpack-cli`, `webpack-dev-server`, and `copy-webpack-plugin`.
  - `scripts`: this section provides two commands
    - `build`: use webpack to bundle the source code into the `./dist`[12] folder.
    - `start`: Start a development server that will bundle the code and serve it right away. It also monitors source code changes and re-bundle if needed.

TODO: Check if we had cargo build in the parent folder before You need to install Webpack and its dependencies by going into the `client` folder, and run `npm install`

[12]This is the default location, so you won't find that mentioned in the code or configuration.

. Once the dependencies are installed, you can run `npm run start`, which will call `webpack-dev-server`. This development server runs webpack to bundle your code whenever your code changes, and serves it on the address `http://localhost:8080` . When you open that URL in a web browser, you should see an alert pop up with the message "Hello, hello-wasm!" (Figure **??**). TODO: Mention error opensslErrorStack: ['error:03000086:digital envelope routines::initialization error'],

Figure 1.1: The pop up alert

The development server, as the name suggests, is for development only. If you want to put this website into production, you'll have to:

- Run `npm run build`.
- Deploy to a production-ready web server the files created in the './dist' folder.

1.4 Resizing image with WebAssembly

The hello world project you just implemented might seem trivial. Why should JavaScript call Wasm, then let Wasm call the JavaScript web API `alert`, instead of letting JavaScript call `alert` directly? Where Wasm truly shines is to replace the performance bottleneck in JavaScript applications. Because Wasm is designed to run at near-native speed, it makes sense to offload performance-critical parts of a JavaScript application to Wasm, while keeping the rest in JavaScript for flexibility and ease of development.

One example of a performance-critical job is image processing in the frontend. Image processing algorithms are usually computationally intensive. If one can use Wasm to handle the core image processing algorithm, it might be able to run much faster than a JavaScript implementation.

You've implemented the cat photo upload service, but it wouldn't be complete without some basic image processing functionality, like resizing and rotation. Therefore, you're going to build a very basic image processing tool using JavaScript and Wasm. Let's start with one of the simplest functionality: resizing.

The simplest way to represent an image on a computer is to store the color value of each pixel. As you might have learned in basic physics class, different colors can be created by adding red, green, and blue lights together at different intensity. If you represent each color component's intensity with an 8-bit integer, you can represent $2^8 \times 2^8 \times 2^8 = 256 \times 256 \times 256 = 1677216$ different colors.

To save storage space, an image can be compressed in many ways so it can be represented more efficiently in memory. There are also hundreds of file formats for storing the image data, like PNG, JPEG, and GIF. Since this is not a book on digital image processing, you are going to rely on an existing Rust crate called `image` to handle all the nitty-gritty of image formats. The `image` crate not only helps you read and write various image formats, it also provides several image processing algorithms like resize, rotate, invert, etc. This also demonstrates one of the benefits of compiling Rust to Wasm: you can build on top of Rust's vibrant crates ecosystem for reliable and high-performance libraries.

First, you need to create a Wasm project using the same command as before:

```
wasm-pack new wasm-image-processing
```

This time, you can name the project `wasm-image-processing`. Then let's add the `image` crate to the `[dependencies]` section in the `Cargo.toml`:

TODO: Update Rust edition TODO: cargo add web_sys

```
[package]
name = "wasm-image-processing"
//...

[dependencies]
wasm-bindgen = "0.2.63"
image = "0.24.1"
```

Let's first think about how the API exposing the JavaScript should look like. The first feature you want to expose to JavaScript is a function that can resize an image. To make it easier, you can make the function shrink the image by half, so you don't have to deal with passing different resize ratios. The function might be something that looks like that in Listing **??**.

```
extern crate web_sys;

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;
```

```rust
#[wasm_bindgen]
pub fn shrink_by_half(
    original_image: SomeKindOfImageType,
    width: u32,
    height: u32
) -> SomeKindOfImageType {
    // ...
}
```

The `shrink_by_half()` function should take an image of some type you don't know yet (`SomeKindOfImageType`), the width and height[13] of that image (as `u32`), and returns a shrunk image.

What should be the type for the `original_image` and the image it returns? you

[13]Although you can avoid passing the width and height parameter and derive those values from the image itself, it's easier to pass them because the functions from the `image` crate needs them.

can maybe take some hint from the `resize` function you'll be using from the `image` crate. The function is located in `image::imageops`, and it's function signature is shown in Listing ??[14].

```
pub fn resize<I: GenericImageView>(
    image: &I,
    nwidth: u32,
    nheight: u32,
    filter: FilterType
) -> ImageBuffer<I::Pixel, Vec<<I::Pixel as Pixel>::Subpixel>>
where
    I::Pixel: 'static,
    <I::Pixel as Pixel>::Subpixel: 'static,
```

The `image` parameter takes an image that implements the `GenericImageView` trait. So you know you need to receive some kind of image data that can be

[14]https://docs.rs/image/0.23.3/image/imageops/fn.resize.html

transformed into a type that implements `GenericImageView`. The return type is an `ImageBuffer`, which can be transformed into something that JavaScript can interpret as an image. It also takes the new width (`nwidth`) and new height (`nheight`) as `u32`. The final parameter `filter` takes an enum `FilterType`. This allows you to select which algorithm to use to scale up the image. You can choose the Nearest Neighbor algorithm[15] for its simplicity and speed.

So now you know that you need something that can be transformed into something that implements the `GenericImageView` trait, maybe you can also see what the frontend can provide. You can create a frontend project inside the current `wasm-image-processing` folder as before:

```
npm init wasm-app client
```

Inside the `client/index.html` file, you can copy and paste the following HTML code (Listing **??**).

[15] https://en.wikipedia.org/wiki/Image_scaling#Nearest-neighbor_interpolation

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Cat image processor</title>
  </head>
  <body>
    <noscript>
      This page contains webassembly and javascript content,
      please enable javascript in your browser.
    </noscript>

    <input type="file"
     name="image-upload"
     id="image-upload"
```

```
     value=""
    >
    <br>
    <button id="shrink">Shrink</button>
    <br>
    <canvas id="preview"></canvas>


    <script src="./bootstrap.js"></script>
  </body>
</html>
```

The page consists of the following elements:

- `<input type="file">`: This is the file selector that allows you to select an image from your computer.

- `<button>Shrink</button>`: When this button is clicked, you should call the

Wasm function to shrink the image.

- `<canvas>`: This canvas is used the display the image.

The `<canvas>` is an HTML element that can be used to draw images with JavaScript. You can render an image onto it using JavaScript APIs. It also provides some APIs to read the rendered image data, which will be handy for converting an image into something Rust/Wasm can understand.

Let's break this process into three steps:

1. Use the `<input type="file">` to load a local image onto the `<canvas>`.
2. Extract the image data from the `<canvas>` and pass it to Wasm for resizing.
3. Receive the resized image data from Wasm, and display it onto the `<canvas>`.

Loading an image file onto the `<canvas>`

You can load the image file onto the `<canvas>` just with JavaScript. Open the `index.js`[16] and add the code in Listing **??**.

[16] It is loaded in `index.html` through `bootstrap.js`, thanks to the template.

```
function setup(event) {
  const fileInput = document.getElementById('image-upload')
  fileInput.addEventListener('change', function(event) {
    const file = event.target.files[0]
    const imageUrl = window.URL.createObjectURL(file)

    const image = new Image()
    image.src = imageUrl

    image.addEventListener('load', (loadEvent) => {
      const canvas = document.getElementById('preview')
      canvas.width = image.naturalWidth
      canvas.height = image.naturalHeight
      canvas.getContext('2d').drawImage(
        image,
```

```
        0,
        0,
        canvas.width,
        canvas.height
      )
    })
  })
}

if (document.readState !== 'loading') {
  setup()
} else {
  window.addEventListener('DOMContentLoaded', setup);
}
```

This piece of code defines a `setup()` function. The function is called immediately if the page is loaded (`document.readyState !== 'loading'`), otherwise, it will be called once the `DOMContentLoaded` event fires.

In the `setup()` function, you monitor the `change` event on the `<input type="file">`. Whenever the user selects a new file with the `<input>`, the `change` will fire. The `<input type="file">` has an attribute `.files`, which returns a list of files you selected as JavaScript `File` objects. you can reach this `FileList` by referencing the `event.target` object (i.e. the `<input type="file">`).

To draw this file onto the `<canvas>`, you need to convert it to an `HTMLImageElement` (JavaScript representation of an `<img>` element). When writing HTML, you set the `src` attribute on the `<img>` element to specify the URL of the image. But the file you just loaded is from a local file system, how can you get an URL for it? The `window.URL.createObjectURL()`[17] method is designed for this. It takes a `File` object as input and returns a temporary URL for it. The URL's lifetime is tied to the `document` in which it was created. Therefore, the following code turns the

[17] https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL

loaded image file into an HTMLImageElement:

```
const file = event.target.files[0]

const imageUrl = window.URL.createObjectURL(file)



const image = new Image()

image.src = imageUrl
```

After you set the src attribute and the file is loaded, a load event will fire. In Listing **??**, you listen for the load event and draws the image onto the canvas. Because you didn't specify the width and height of the <canvas> element in HTML, it has a default of 300 x 150 pixels size. But the image might have a different size, so you can set the canvas's width and height to the naturalWidth and naturalHeight of the HTMLImageElement. These two values represent the intrinsic size of the image.

Finally, you can draw the image onto the <canvas>. But you can't draw directly to the HTMLCanvasElement (i.e., the return value of document.getElementById `('preview'))`. You'll need to first get a 2D drawing context by calling `canvas.getContext('2d')`. Only after that can you call the `.drawImage()` function on that context. The `drawImage()` function can take 3 arguments:

- `image`: the `HTMLImageElement` you created from the file.
- `dx`: the x-axis coordinate of the top-left corner of the image's position.
- `dy`: the y-ayis coordinate of the top-left corner of the image's position.

Both `dx` and `dy` are set to 0 so the image's top-left corner matches the canvas's top-left corner.

TODO: Add the dependencies in package.json:

```
{
  "name": "create-wasm-app",
  // ...
  "dependencies": {
    "wasm-image-processing": "file:../pkg"
```

```
  },
  "devDependencies": {
    // remove hello-wasm-pack
    "webpack": "^4.29.3",
    "webpack-cli": "^3.1.0",
    "webpack-dev-server": "^3.1.5",
    "copy-webpack-plugin": "^5.0.0"
  }
}
```

TODO: Explain the code above TODO: Must run npm install AFTER updating package.json

To test this code, run `wasm-pack build` in the `wasm-image-processing` folder, this generates the Wasm module for the client to consume. Then run `npm install` followed by `npm run start` inside the `wasm-image-processing/`

`client` directory, the pre-configured `webpack-dev-server` will start running. You can open a browser and visit `http://localhost:8080` to see the page in action (Figure **??**).

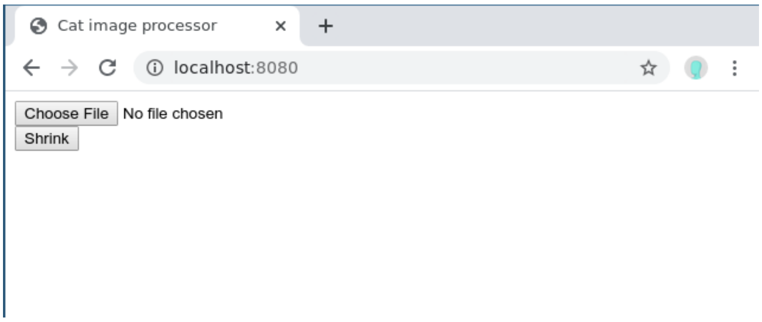

Figure 1.2: Loading a local image onto the `<canvas>`

Passing the image to Wasm

Now the images can be loaded onto the `<canvas>`, but what kind of data format can the canvas represent? As mentioned before, images can be represented as a collection of pixels; each pixel's color can be represented by integers. Therefore, an integer array can be a good fit because both JavaScript and Rust can easily handle it.

As mentioned before, a common way to represent an image is to store each pixel as four numbers:

- R: the intensity of the red channel
- G: the intensity of the green channel
- B: the intensity of the blue channel
- A: the Alpha channel, meaning the transparency of the pixel. Alpha of 0% means totally transparent, and Alpha of 100% means totally opaque

If a `u8` represents each value, then it can range between 0 and 255. On the Rust side, this can be represented by a `Vec<u8>`. On the JavaScript side, it can be represented by a `Uint8ClampedArray`[18]

On the Rust side, you can now complete the function definition, updating the `lib.rs` file as in Listing **??**.

[18] The term "clamped" in the name means the value is "clamped" to the range from 0 to 255. If you set a value larger then 255 it will become 255, and if you set a negative number it will become 0.

```
extern crate web_sys;

mod utils;

use image::{RgbaImage};
use image::imageops;

use wasm_bindgen::prelude::*;

// ... wee_alloc setup

#[wasm_bindgen]
pub fn shrink_by_half(
    original_image: Vec<u8>,
    width: u32,
```

```
    height: u32
) -> Vec<u8> {
    let image: RgbaImage =
        image::ImageBuffer::from_vec(
            width, height, original_image
        ).unwrap();
    let output_image = imageops::resize(
        &image,
        width / 2,
        height / 2,
        imageops::FilterType::Nearest
    );

    output_image.into_vec()
}
```

The `original_image` parameter is an 1-D `Vec<u8>`. To reconstruct an 2-D image from an 1-D array, you need to also pass the width and height[19] You can use the `image::ImageBuffer::from_vec()` function to turn the `Vec<u8>` back into an `RgbaImage`. Because the `RgbaImage` type implements the `GenericImageView` trait, you can pass this `RgbaImage` to `imageops::resize` to resize the image. Once you received the resized image, it can then be turned back into a `Vec<u8>` with `.into_vec()` and returned to JavaScript.

On the frontend page, you can add an event listener on the "Shrink" button, so it triggers a call to `shrink_by_half()` Wasm function, so setting the `index.js` file as in Listing **??**.

```
import * as wasmImage from "wasm-image-processing"


function setup(event) {
```

[19] In theory you only need to pass either the width or the height, because the other one can be calculated from the size of the array and the specified dimension. But in the example you pass both so the code is simpler.

```
  // ...
  //
  const shrinkButton = document.getElementById('shrink')
  shrinkButton.addEventListener('click', function(event) {
      const canvas = document.getElementById('preview')
      const canvasContext = canvas.getContext('2d')
      const imageBuffer = canvasContext.getImageData(
          0, 0, canvas.width, canvas.height
      ).data

      const outputBuffer = wasmImage.shrink_by_half(
          imageBuffer, canvas.width, canvas.height
      )

      const u8OutputBuffer = new ImageData(
```

```
          new Uint8ClampedArray(outputBuffer), canvas.width / 2
      )

      canvasContext.clearRect(
          0, 0, canvas.width, canvas.height
      );
      canvas.width = canvas.width / 2
      canvas.height = canvas.height / 2
      canvasContext.putImageData(u8OutputBuffer, 0, 0)
  })
}

if (document.readState !== 'loading') {
  setup()
} else {
```

```
  window.addEventListener('DOMContentLoaded', setup);
}
```

Notice that you imported the wasm-image-processing, which is the crate in the top-level folder. When the button is clicked, you need to first get the 2-D context from the canvas. The context exposes a function getImageData, which can retrieve part of the canvas as an ImageData object. The first two parameters specify the X and Y coordinate of the top right corner of the area you want to retrieve. The next two parameters specify the width and height of that area. Here you get the whole canvas. The ImageData has a read-only data attribute which contains the Uint8ClampedArray representation of the RGBA values.

You can pass this Uint8ClampedArray to the wasmImage.shrink_by_half() Wasm function imported at the beginning of the file. The return value will be a Vec<u8> representation of the shrunken image. You can convert it back to Uint8ClampedArray and wrap it in an ImageData.

To show this shrunken image on the <canvas>, you can follow the three steps shown in the code:

1. Clear the canvas with `clearRect()`.
2. Set the canvas size to the new shrunken size.
3. Draw the new `ImageData` onto the <canvas> using `putImageData()`.

To test this application, follow these steps:

1. In the `wasm-image-processing` folder, run `wasm-pack build`. This compiles the Rust code into Wasm, located in the `pkg` folder.
2. Move into the `client` folder, run `npm install && npm run start`.
3. Open a browser, go to `http://localhost:8080` (Figure **??**)
4. Click the "Choose File" button. A file selector window will pop up. Select an image file (PNG) from your computer (Figure **??**).
5. Click the "Shrink" button. Figure **??**

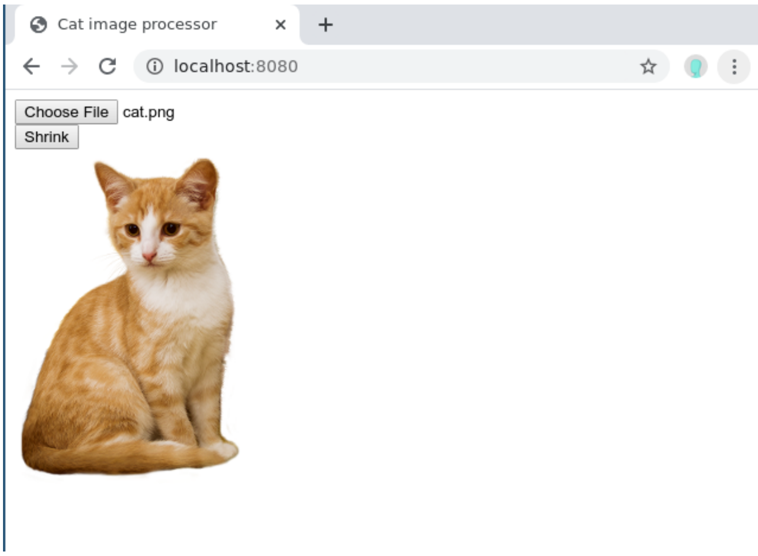

Figure 1.3: File selected

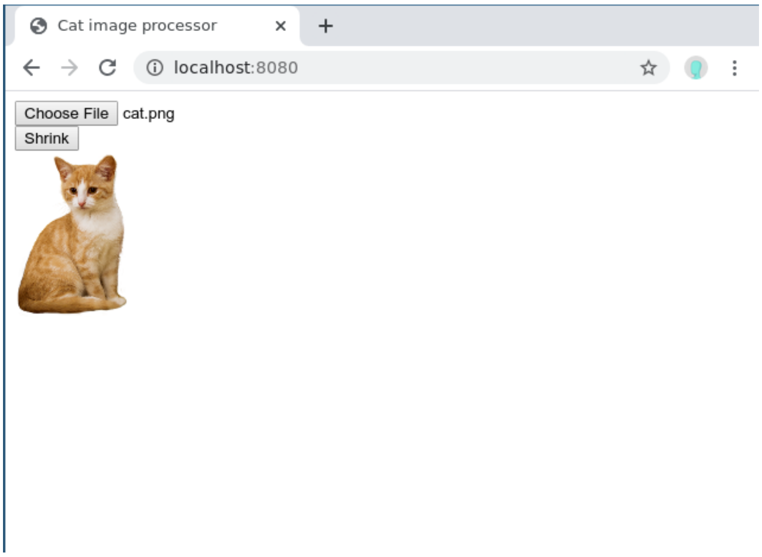

Figure 1.4: After clicking the shrink button

■ **NOTE** The method shown in this section is not the most efficient way. As a rule of thumb, you want to avoid unnecessary copying between JavaScript memory and the WebAssembly linear memory. Quoting from the official *Rust and WebAssembly* book[20]:

..., a good JavaScriptWebAssembly 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.

Therefore, you might want to try loading the image directly in Rust/Wasm like this great open source project demonstrates: `https://www.imagproc.com/main`.

Another potential improvement is that you can offload the computation to a Web Worker. Currently, our JavaScript code calls the image processing function on the main event loop. While the image processing function is running, it might block further user

[20] https://rustwasm.github.io/book/game-of-life/implementing.html

interaction. Web Worker is a web technology that allows you to run scripts in the background thread so that it won't block the user interface. You can also find an example of a Web Worker in the `www.imageproc.com` code.

1.5 Writing the whole frontend in Rust

Up until now, you've been building a web page in JavaScript and call Wasm functions when needed. But is it possible to write everything in Rust? The answer is yes, but it relies on a programming pattern called the *Virtual DOM*.

The Virtual DOM is a concept popularized by the popular JavaScript framework React[21]. When you build a web page in plain JavaScript and need to change something on the screen, you need to call many DOM APIs *imperatively*. That means you need to say, "Get me this `<p>` element and change its text to 'foobar', then get that button and turn it red.". But when the page grows more and more complicated, this approach might lead to chaos and human errors. Instead, React uses a *declarative* approach. You instead say, "I

[21] https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom

want this `<p>` to contain ’foobar’, and I want the button to be red", and React needs to figure out how to get the page from the current state to your desired state.

Whenever the desired state changes, React will "render" the page to a Virtual DOM, which is an in-memory representation of the real DOM. The Virtual DOM can figure out which parts of the page changed comparing to the previous state, and call DOM API to update (or *reconcile* in React terminology) only the required part of the real DOM. This allows the developer to focus on the overall UI declaration instead of worrying about which part of the DOM to update.

If you build a Virtual DOM in Rust and compile it to Wasm, you can write the rest of the page in Rust, which interacts with the Virtual DOM. Then the Virtual DOM uses crates like `web-sys` to interact with the real DOM API to reconcile the difference. There have been many attempts. you’ll introduce one of the most popular framework called Yew[22]. Yew is heavily influenced by the design of React and Elm[23].

[22] https://yew.rs/docs/

[23] Another popular web framework/language for building frontend applications

TODO: Everything below needs to be rewritten

Setting up Yew

First, let's set up a minimal project with Yew and take a look at a hello world project. Yew provides a project template just like `wasm-pack`, so you can easily set up the project. To start, run the following command to create a project using the `yew-wasm-pack-template`.

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

When the command-line tool asks you for a project name, you can name it `yew-image-processing`.

■ **NOTE** You might see an error message on "Error replacing placeholders". This is a known issue with `yew-wasm-pack-template`, and it won't affect the functionality. It might be fixed in future versions. The project folder will still be created and you can safely continue.

■ **NOTE** Yew is very flexible with tooling. You can choose which Wasm build tool to use inside Yew. The options are:

- wasm-pack
- wasm-bindgen
- cargo-web

You can also choose which Rust-and-Web-API bindings crate you want to use:

- web-sys
- stdweb

In this chapter, you'll stick with the tools and crates maintained by the Rust/Wasm Working Group, which are `wasm-pack` and `web-sys`.

The project already contains configurations for `wasm-pack` and `webpack`, which you are already familiar with from the wasm-pack template. The `README` documentation suggests using the Yarn package manager[24], which is an alternative to npm. But both

[24] https://yarnpkg.com/

package manager accepts the same `package.json` format, so you can still use npm.

The template also includes a TodoMVC[25] example. You can simply run the following command to start it:

```
npm install && npm run start:dev
```

A webpack development server will start on port 8000. You can open a browser and go to `http://localhost:8000` to play with the example.

1.6 A hello world example

But the TodoMVC is too complicated as a hello world example. Let's simplify the example using the following steps:

1. Replace the content of `src/app.rs` with Listing **??**.

2. Rename the `todomvc.js` in `webpack.config.js` to `yew-image-`

[25] TodoMVC is a project which implements the same to-do list application in multiple frontend frameworks. Its purpose is to help developers see how different frontend frameworks compare.

processing.js.

3. Rename the `todomvc.wasm` in `webpack.config.js` to `yew-image-processing.wasm` (Listing ??).

4. Include `yew-image-processing.js` in `static/index.html` instead of `todomvc.js`.

5. Remove the TodoMVC CSS stylesheets in `static/index.html` (Listing ??).

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

pub struct App {
    link: ComponentLink<Self>,
    value: i64,
}

pub enum Msg {
```

```
    AddOne,
}

impl Component for App {
    type Message = Msg;
    type Properties = ();
    fn create(
        _: Self::Properties,
        link: ComponentLink<Self>,
    ) -> Self {
        Self { link, value: 0 }
    }

    fn change(&mut self, _props: Self::Properties) -> 
      ↪ ShouldRender {
```

```
        false
    }

    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::AddOne => self.value += 1,
        }
        true
    }

    fn view(&self) -> Html {
        html! {
            <div>
                <button
                 onclick=self.link.callback(|_| Msg::AddOne)
```

```
                >
                    { "+1" }
                </button>
                <p>{ self.value }</p>
            </div>
        }
    }
}
```

```
const path = require('path');
const WasmPackPlugin = require('@wasm-tool/wasm-pack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');


const distPath = path.resolve(__dirname, "dist");
module.exports = (env, argv) => {
  return {
```

```
    devServer: {
      contentBase: distPath,
      compress: argv.mode === 'production',
      port: 8000
    },
    entry: './bootstrap.js',
    output: {
      path: distPath,
      filename: "yew-image-processing.js",
      webassemblyModuleFilename: "yew-image-processing.wasm"
    },
    module: {
      rules: [
        {
          test: /\.s[ac]ss$/i,
```

```
          use: [
            'style-loader',
            'css-loader',
            'sass-loader',
          ],
        },
      ],
    },
    plugins: [
      new CopyWebpackPlugin([
        { from: './static', to: distPath }
      ]),
      new WasmPackPlugin({
        crateDirectory: ".",
        extraArgs: "--no-typescript",
```

```
      })
    ],
    watch: argv.mode !== 'production'
  };
};
```

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Yew image processing</title>
        <!-- Stylesheets removed -->
    </head>
    <body>
        <!-- JS file renamed -->
        <script src="/yew-image-processing.js"></script>
```

```
    </body>
</html>
```

.

Now you can run `npm install`, followed by `npm run start:dev`, and refresh your browser to test it. You don't need to explicitly run `wasm-pack build`, because when you run `npm run start:dev`, the command triggers Webpack. In the Webpack configuration (Listing **??**), there is a `WasmPackPlugin` configured so it will run `wasm-pack build` for you.

In a production build, Webpack will utilize `wasm-pack` to compile `src/app.rs` file and other boilerplate Rust files to a Wasm module. It then will package other boilerplate JavaScript files into `yew-image-processing.js`. The `index.html` file then loads the `yew-image-processing.js`, which then imports `yew-image-processing.wasm` and runs the Yew app.

To understand how this example works, you first need to understand the Elm architecture[26], which influences Yew.

The Elm architecture consists of three core concepts:

- Model: the state of the application.
- View: a way to turn the state into the UI (HTML).
- Update: a way to update the state based on the message (Msg) triggered by user interaction on the UI.

Their interactions are illustrated in Figure **??**. Yew loosely follows this architecture.

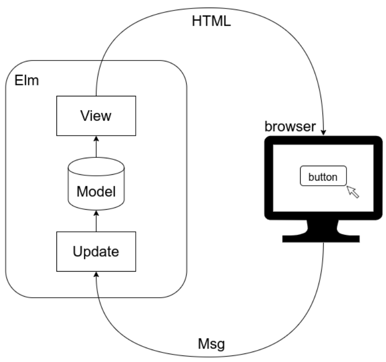

Figure 1.5: Elm architecture

The hello world example has a counter as its Model. The counter is incremented

[26] https://guide.elm-lang.org/architecture/

whenever the user clicks a "+1" button in the browser. The counter is also shown on the page, so the number updates whenever the Model changes (Figure **??**).

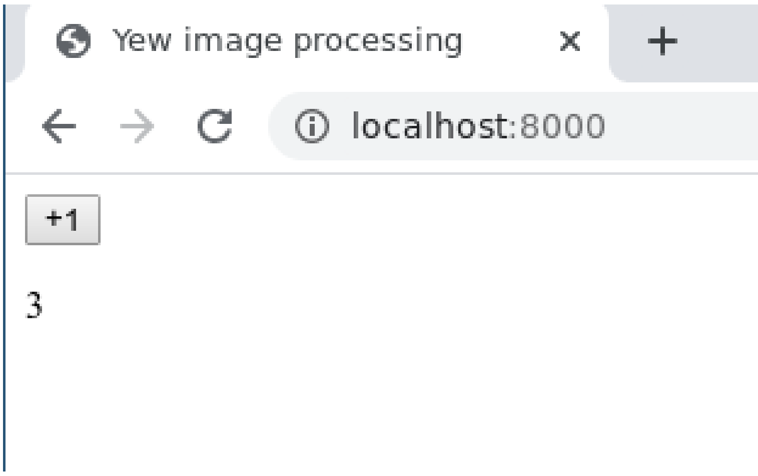

Figure 1.6: The hello world Yew application

The core of this example is in `src/app.rs` (Listing **??**), which defines a component called `App`. But before you dive into its details, let's first understand how it's loaded in the page. In `webpack.config.js` (Listing **??**), you see the `entry` field is `./bootstrap.js`. This means the entry point of this web page is the `bootstrap.js` file (Listing **??**). The `bootstrap.js` file simply loads the compiled Wasm module (located in `./pkg`) and call the `module.run_app()` function.

```
import("./pkg").then(module => {
  module.run_app();
});
```

.

The `run_app()` function is defined in `src/lib.rs` (Listing **??**). The most important line in that function is

```
yew::start_app::<app::App>();
```

This line starts the yew application by mounting the `app::App` component onto the `<body>` of the HTML page.

```
#![recursion_limit = "512"]


mod app;


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;

// This is the entry point for the web app
#[wasm_bindgen]
pub fn run_app() -> Result<(), JsValue> {
    wasm_logger::init(wasm_logger::Config::default());
    yew::start_app::<app::App>();
    Ok(())
}
```

Finally, you can come back to `src/app.rs`(Listing **??**). The file first declares a `struct` called `App`, which is the only component rendered to the screen. This `struct` contains a Model called `value`. This Model is the counter on how many times the button is clicked.

you also implement the `Component` trait on `App`. The first function `create()` takes care of the initialization of the component. As you can see `value` is initialized as `0`. o The `view()` function is the key to turn the model into HTML. The `html!{}` macro allows you to write HTML syntax inside Rust, similar to JSX[27] in React. This `view()` function defines the HTML that will render a <div> containing a <button> and a <p>. Notice that the text inside the <p> is not hardcoded, but it refers to a variable `self.value`, wrapped inside a pair of curly brackets. This tells Yew to substitute the text with the value of `self.value` when `view()` is called. So whenever `value` changes, `view()` will be called, and the Virtual DOM will reconcile the change to the DOM and show it on screen.

[27] https://reactjs.org/docs/glossary.html#jsx

How do you update the state? In Yew, you can update the state by sending messages to the component. In the same file, you defined a message `enum Msg`, which has only one variant called `AddOne`. The `update()` function on the `App` component handles incoming messages and updates the state accordingly. In this example, a `Msg::AddOne` message will increment the `self.value` model.

To send the message when the button is clicked, you need to utilize the `ComponentLink` mechanism. `ComponentLink` is a way to register a callback, that will send the message to the component's `update` method. As you can see you added a `link: ComponentLink<Self>` field in the `App struct`. In the `onclick` event handler of `button`, you call `self.link.callback(|_| Msg::AddOne)`.

1.7 Re-implement the image processing frontend with Yew

You can also re-implement the `client` part of the `wasm-image-processing` project in Yew. The process is quite straightforward:

1. Create a Yew component.
2. Move the HTML page into the `view()` function of the component.
3. When the buttons are clicked, send `Msg` instead of calling JavaScript directly.
4. Convert the JavaScript button `onclick` handlers to Rust code using `web-sys`.

So first, let's clean up `src/app.rs` to become a skeleton Yew component, then add the `view()` function to render the HTML Listing **??**

```
use image::imageops;
use image::RgbaImage;
```

```
use std::rc::Rc;
use wasm_bindgen::prelude::*;
use wasm_bindgen::{Clamped, JsCast};
use yew::services::reader::File;
use yew::{
    html, ChangeData, Component, ComponentLink, Html,
    ShouldRender,
};

pub struct App {
    link: ComponentLink<Self>,
}

pub enum Msg {
    // ...
```

```
}

impl Component for App {
    type Message = Msg;
    type Properties = ();
    fn create(
        _: Self::Properties,
        link: ComponentLink<Self>,
    ) -> Self {
        Self { link }
    }

    fn change(&mut self, msg: Self::Message) -> ShouldRender {
        false
    }
```

```
    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        // ...
    }

    fn view(&self) -> Html {
        html! {
            <div>
                <input type="file"
                 name="image-upload"
                 id="image-upload"
                 value=""
                 onchange={ /* ... */ }
                />
                <br />
```

```
                <button id="shrink" onclick={ /* ... */ }>
                    { "Shrink" }
                </button>
                <br />
                <canvas id="preview"></canvas>
            </div>
        }
    }
}
```

In the `App` component's `view()` function, you use the `html!{}` macro to render the HTML. The `html!{}` macro is similar to JSX in React, which allows you to write HTML syntax in another language. However, the `html!{}` is stricter in terms of syntax then most major browser's HTML implementation, so you need to remember to properly close HTML tags as XML tags (e.g., `<br />` instead of just `<br>`).

Once the HTML is in place, you can start to migrate the JavaScript code to Rust.

Let's start with loading an image file onto the canvas. As in the hello world example, you can attach a `ComponentLink::callback()` on the `<input type="file">`'s `onchange` handler. The callback should send a message to the `update()` function, which should then load the image and show it on the canvas. The outline of this flow should look like Listing **??**. Notice that in the `onchange` handler, the event contains a `FileList` object, and so you use `js-sys` to convert it into a `Vec<File>` for ease of processing.

```
use image::imageops;
use image::RgbaImage;
use std::rc::Rc;
use wasm_bindgen::prelude::*;
use wasm_bindgen::{Clamped, JsCast};
use yew::services::reader::File;
use yew::{
    html, ChangeData, Component, ComponentLink, Html,
```

```
    ShouldRender,
};


pub struct App {
    link: ComponentLink<Self>,
}


pub enum Msg {
    LoadFile(Vec<File>),
    // ...
}


impl Component for App {
    type Message = Msg;
    type Properties = ();
```

```
    fn create(
        _: Self::Properties,
        link: ComponentLink<Self>,
    ) -> Self {
        Self { link }
    }

    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::LoadFile(files) => {
                // ... Load the file onto the canvas
            }
        }
        true
    }
```

```
    fn view(&self) -> Html {
        html! {
            <div>
                <input type="file"
                 name="image-upload"
                 id="image-upload"
                 value=""
                 onchange=self.link.callback(move |value| {
                    let mut result = Vec::new();
                    if let ChangeData::Files(files) = value {
                        let files = js_sys::try_iter(&files)
                            .unwrap()
                            .unwrap()
                            .into_iter()
```

```
                            .map(|v| File::from(v.unwrap()));
                        result.extend(files);
                    }
                    Msg::LoadFile(result)
                }) />
                <br />
                <button id="shrink" onclick={ /* ... */ }>
                    { "Shrink" }
                </button>
                <br / >
                <canvas id="preview"></canvas>
            </div>
        }
    }
}
```

In the `update()` function, if you receive the `Msg::LoadFile` message, you need to do what Listing `lst:show-image` does, but in Rust. You can convert all the JavaScript into Rust with the help from `web-sys`. `Web-sys` is a crate that defines the binding to Web APIs and Rust. The Rust code is shown in Listing **??**.

```
impl Component for App {
    // ...
    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::LoadFile(files) => {
                let file = &files[0];
                let file_url =
                    web_sys::Url::create_object_url_with_blob(
                        &file,
                    )
                    .unwrap();
```

```
                let document = web_sys::window()
                    .unwrap()
                    .document()
                    .unwrap();
                let image = Rc::new(
                    document
                        .create_element("img")
                        .unwrap()
                        .dyn_into::<web_sys::HtmlImageElement>()
                        .unwrap(),
                );
                image.set_src(&file_url);

                let image_clone = image.clone();
```

```
                let callback = Closure::wrap(Box::new(
                    move || {
                        let canvas = document
                            .get_element_by_id("preview")
                            .unwrap();
                        let canvas: web_sys::HtmlCanvasElement =
                            canvas
                                .dyn_into::<web_sys::HtmlCanvasElement>()
                                .map_err(|_| ())
                                .unwrap();

                        let context = canvas
                            .get_context("2d")
                            .unwrap()
```

```
                            .unwrap()
                            .dyn_into::<web_sys::CanvasRenderingContext2d>()
                            .unwrap();
                        canvas.set_width(
                            image_clone.natural_width(),
                        );
                        canvas.set_height(
                            image_clone.natural_height(),
                        );
                        context
                            .draw_image_with_html_image_element(
                                &image_clone,
                                0.0,
                                0.0,
```

```
                            )
                            .unwrap();
                    },
                )
                    as Box<dyn Fn()>);
                image.set_onload(Some(
                    callback.as_ref().unchecked_ref(),
                ));
                callback.forget();
            }
        }
        true
    }
    // ...
}
```

One important thing to point out is that the `Image`'s `onload` handler takes a JavaScript function as a callback. To define that in Rust, you need to use a `Closure`. Because the `image` needs to be moved into the closure, but you still need to reference it after defining the closure (when you call `image.set_onload()`), the `image` needs to be wrapped in an `Rc` so it can have shared ownership. Also, because the callback might be called after the `update()` finishes, you need to tell Rust not to drop the callback when it goes out of scope (i.e. when the `update()` function finishes). Therefore, you call `callback.forget()` at the end of the function.

Because `web-sys` contains a lot of Web APIs, `web-sys` puts each Web API behind feature flags. You should only enable features that you actually use, so you don't waste time compiling Web APIs you don't need. For this example, you need to add the following features in your `Cargo.toml` file (Listing **??**).

```
[package]
// ...
```

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

[dependencies]
log = "0.4"
strum = "0.17"
strum_macros = "0.17"
serde = "1"
serde_derive = "1"
wasm-bindgen = "0.2.58"
wasm-logger = "0.2"
wee_alloc = { version = "0.4.4", optional = true }
yew = { version = "0.17", features = ["web_sys"] }
image = "0.23.10"
```

```
js-sys = "0.3.45"

[dev-dependencies]
wasm-bindgen-test = "0.3"

[dependencies.web-sys]
version = "0.3.4"
features = [
  'KeyboardEvent',
  'HtmlImageElement',
  'Element',
  'Document',
  'Element',
  'EventTarget',
  'HtmlCanvasElement',
```

```
  'HtmlElement',
  'MouseEvent',
  'Node',
  'Window',
  'CanvasRenderingContext2d',
  'ImageData',
]
```

Finally, you can convert the Shrink button code from JavaScript to Rust as well (Listing **??**).

```
// ...


pub enum Msg {
    LoadFile(Vec<File>),
    Shrink,
}
```

```
impl Component for App {
    // ...

    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::LoadFile(files) => {
                // ...
            }
            Msg::Shrink => {
                let document = web_sys::window()
                    .unwrap()
                    .document()
                    .unwrap();
                let canvas = document
```

```
                    .get_element_by_id("preview")
                    .unwrap();
                let canvas: web_sys::HtmlCanvasElement = canvas
                    .dyn_into::<web_sys::HtmlCanvasElement>()
                    .map_err(|_| ())
                    .unwrap();

                let context = canvas
                    .get_context("2d")
                    .unwrap()
                    .unwrap()
                    .dyn_into::<web_sys::
    ↪ CanvasRenderingContext2d>()
                    .unwrap();
                let width: u32 = canvas.width();
```

```
                let height: u32 = canvas.height();
                let image_buffer = context
                    .get_image_data(
                        0.0,
                        0.0,
                        width.into(),
                        height.into(),
                    )
                    .unwrap()
                    .data();
                let image: RgbaImage =
                    image::ImageBuffer::from_vec(
                        width,
                        height,
                        image_buffer.to_vec(),
```

```
                    )
                    .unwrap();
                let output_image = imageops::resize(
                    &image,
                    width / 2,
                    height / 2,
                    imageops::FilterType::Nearest,
                );
                let output_image_data =
                    web_sys::ImageData::
    ↪ new_with_u8_clamped_array(
                        Clamped(&mut output_image.into_vec()),
                        width / 2
                    ).unwrap();
                context.clear_rect(
```

```
                    0.0,
                    0.0,
                    width.into(),
                    height.into(),
                );
                canvas.set_width(width / 2);
                canvas.set_height(height / 2);
                context
                    .put_image_data(
                        &output_image_data,
                        0.0,
                        0.0
                    )
                    .unwrap();
            }
```

```
        }
        true
    }

    fn view(&self) -> Html {
        html! {
            <div>
                <input type="file"
                 name="image-upload"
                 id="image-upload"
                 value=""
                 onchange=self.link.callback(move |value| {
                    let mut result = Vec::new();
                    if let ChangeData::Files(files) = value {
                        let files = js_sys::try_iter(&files)
```

```
                            .unwrap()
                            .unwrap()
                            .into_iter()
                            .map(|v| File::from(v.unwrap()));
                        result.extend(files);
                    }
                    Msg::LoadFile(result)
                }) />
                <br />
                <button id="shrink"
                 onclick=self.link.callback(move |_| {
                     Msg::Shrink
                 })
                >
                    { "Shrink" }
```

```
                </button>
                <br / >
                <canvas id="preview"></canvas>
            </div>
        }
    }
}
```

Notice that you no longer need to export a Rust function to JavaScript. Everything is in Rust now, so once you've read the image data from the `<canvas>` and do the proper conversion, you can directly call `image::imageops::resize()`.

1.8 Other alternatives

WebAssembly is a versatile platform for many applications, so there are many different tools and frameworks that focus on different topics.

The tools introduced in this chapter are mostly maintained by the Rust and WebAssembly Working Group[28]. That includes the `web-sys` and `js-sys` crates. But `web-sys` provides a very low-level API, which might not be user friendly. Their APIs are also a direct mapping to JavaScript APIs, so the syntax is not idiomatic Rust. There is an alternative implementation for Web APIs called `stdweb`[29]. It provides a higher-level binding between Rust and Web APIs. It also uses a different build system called `cargo -web`[30], which doesn't rely on `npm` and `web-pack` like `wasm-bindgen`. `Stdweb` starts to have `wasm-bindgen` compatibility since version 0.4.16. You can start using `stdweb` in `wasm-bindgen`-based projects, and it can be built using `wasm-bindgen` tooling.

There has also been an effort from the Rust and WebAssembly Working Group to build a high-level toolkit, called `gloo`[31]. However, the toolkit development seems to be less active recently.

There are also many frontend frameworks similar to Yew. They are mostly inspired by

[28]https://github.com/rustwasm/team

[29]https://github.com/koute/stdweb

[30]https://github.com/koute/cargo-web

[31]https://github.com/rustwasm/gloo

popular frontend frameworks and patterns in other languages, like Elm, React and Redux.

Just to name a few (in alphabetical order):

- Darco[32]: Inspired by Elm and Redux.
- Percy[33]: Supports isomorphic web application, meaning the same code runs both on the server side and client side.
- Seed[34]: Inspired by Elm, React and Redux
- Smithy[35]
- rust-dominator[36]
- squark[37]
- willow[38]: Inspired by Elm

[32] https://github.com/utkarshkukreti/draco

[33] https://github.com/chinedufn/percy

[34] https://seed-rs.org/

[35] https://github.com/rbalicki2/smithy

[36] https://github.com/Pauan/rust-dominator

[37] https://github.com/rail44/squark

[38] https://github.com/sindreij/willow

But WebAssembly is not limited to the browser only. In theory, the Wasm runtime can be embedded (or run stand-alone) almost everywhere. Some interesting examples include:

- Serve as backend web servers
- Power Istio[39] plugins
- Run on Internet of Things devices
- Drive robots

There is a cross-industry alliance called the Bytecode Alliance[40] that is driving the development of WebAssembly foundation outside of the browser. Their projects include:

- Wasmtime[41]: a Wasm runtime
- Cranelift[42]: A code generator that powers Wasmtime.

[39] Istio is a service mesh, which allows you to control, manage and observe the network traffic between a network of microserivces

[40] https://bytecodealliance.org/

[41] https://wasmtime.dev/

[42] https://github.com/bytecodealliance/wasmtime/tree/master/cranelift

- Lucet[43]: A Wasm compiler and runtime that allows you to execute untrusted Wasm code in a sandbox
- WAMR[44]: WebAssembly Micro Runtime

Many of these projects are built with Rust or work with Rust. So if you are interested in the development of WebAssembly, you should keep a close eye on their development.

[43] https://github.com/bytecodealliance/lucet/

[44] https://github.com/bytecodealliance/wasm-micro-runtime