Learnable Programming with Rust

What is learnable programming?

Design principles to improve understanding

Show the state of a program

Lowering barriers

Tokio
Learn
API Docs
Blog
aws

Build reliable network applications without compromising speed.

Tokio is an asynchronous runtime for the Rust programming language. It provides the building blocks needed for writing network applications. It gives the flexibility to target a wide range of systems, from large servers with dozens of cores to small embedded devices.

Get Started

Built by the community, for the community.

Hello Tokio

We will get started by writing a very basic Tokio application. It will connect to the Mini-Redis server, set the value of the key `hello` to `world`. It will then read back the key. This will be done using the Mini-Redis client library.

The code

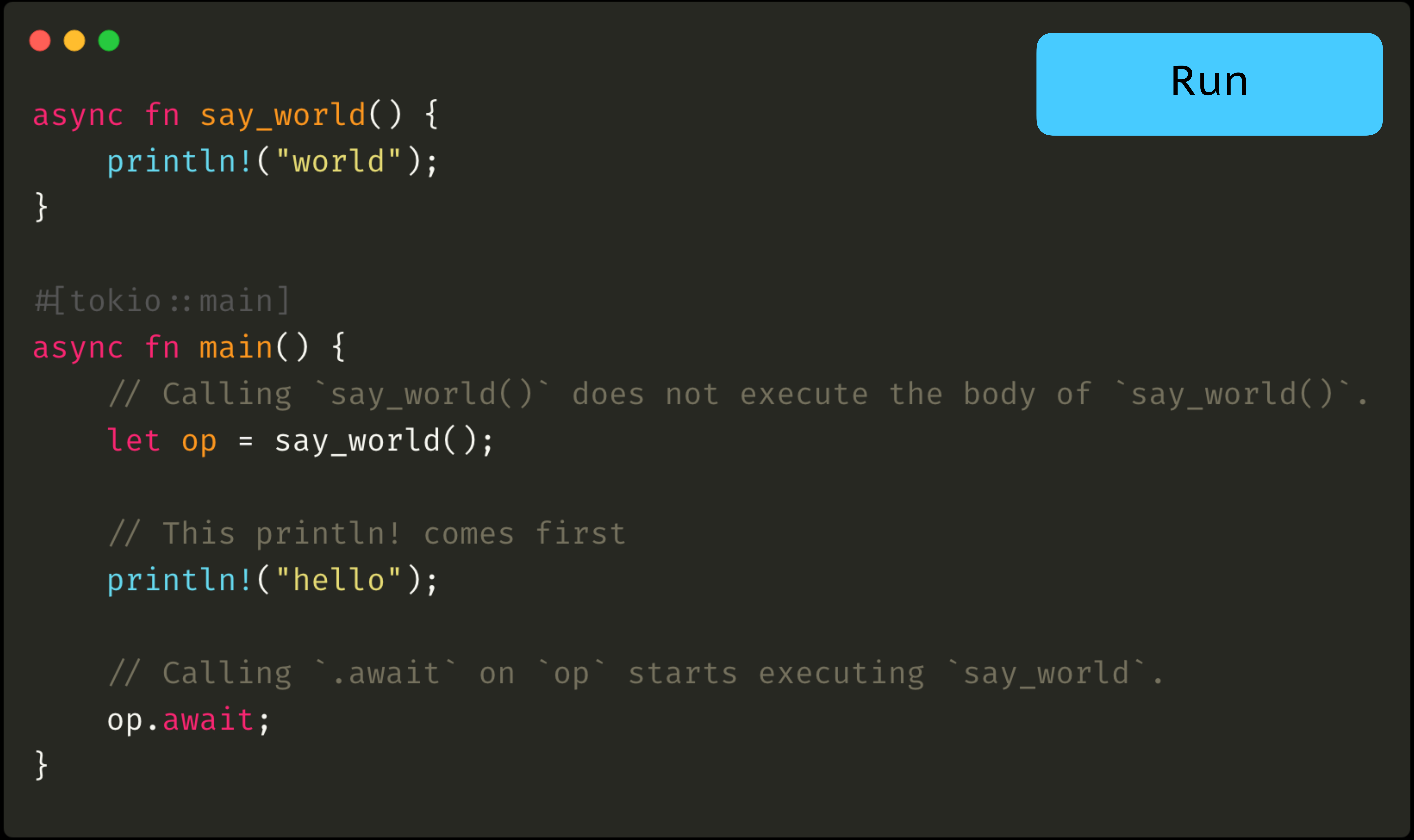
Generate a new crate

Let's start by generating a new Rust app:

```
$ cargo new my-redis
$ cd my-redis
```

Add dependencies

How it works

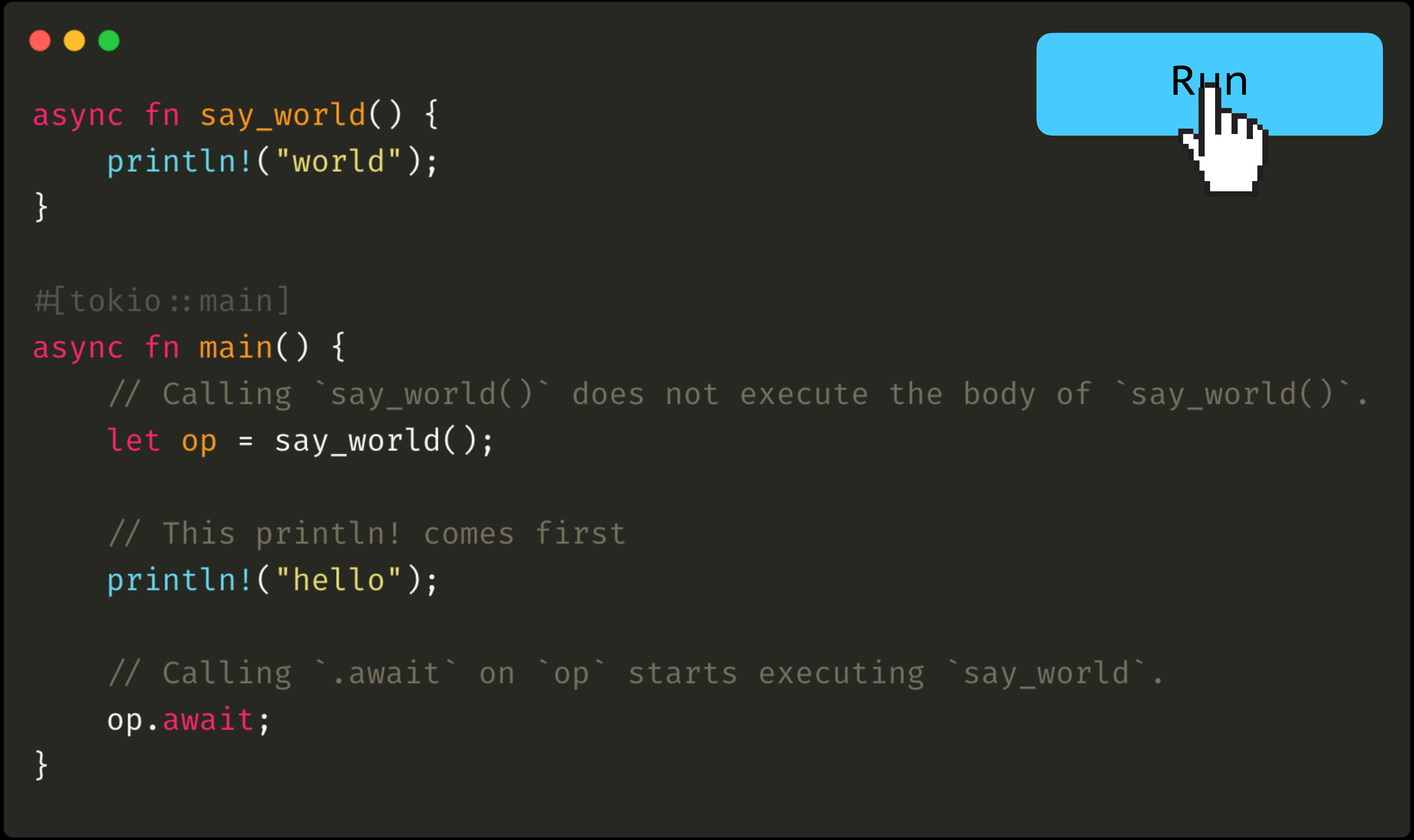


```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```

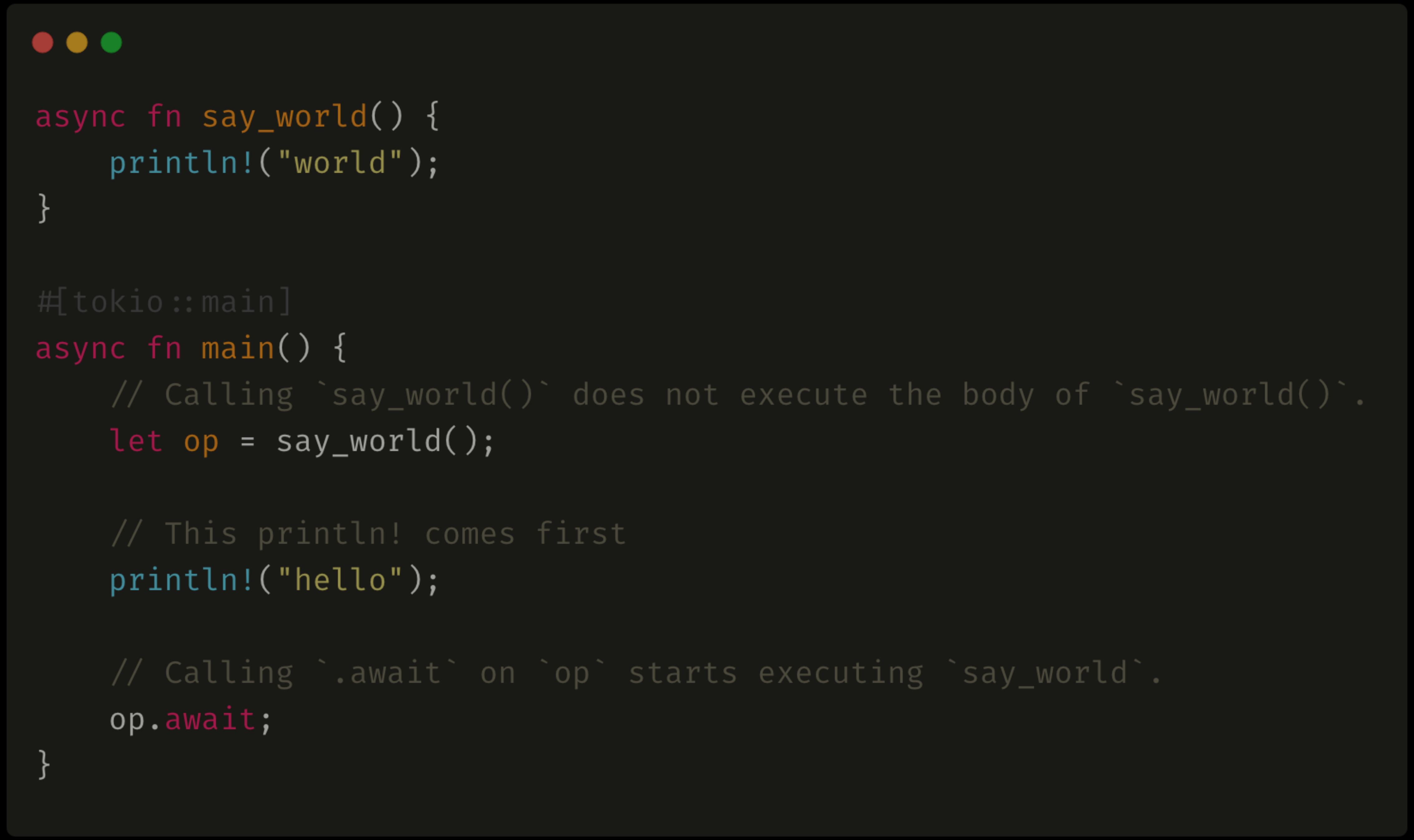


```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```

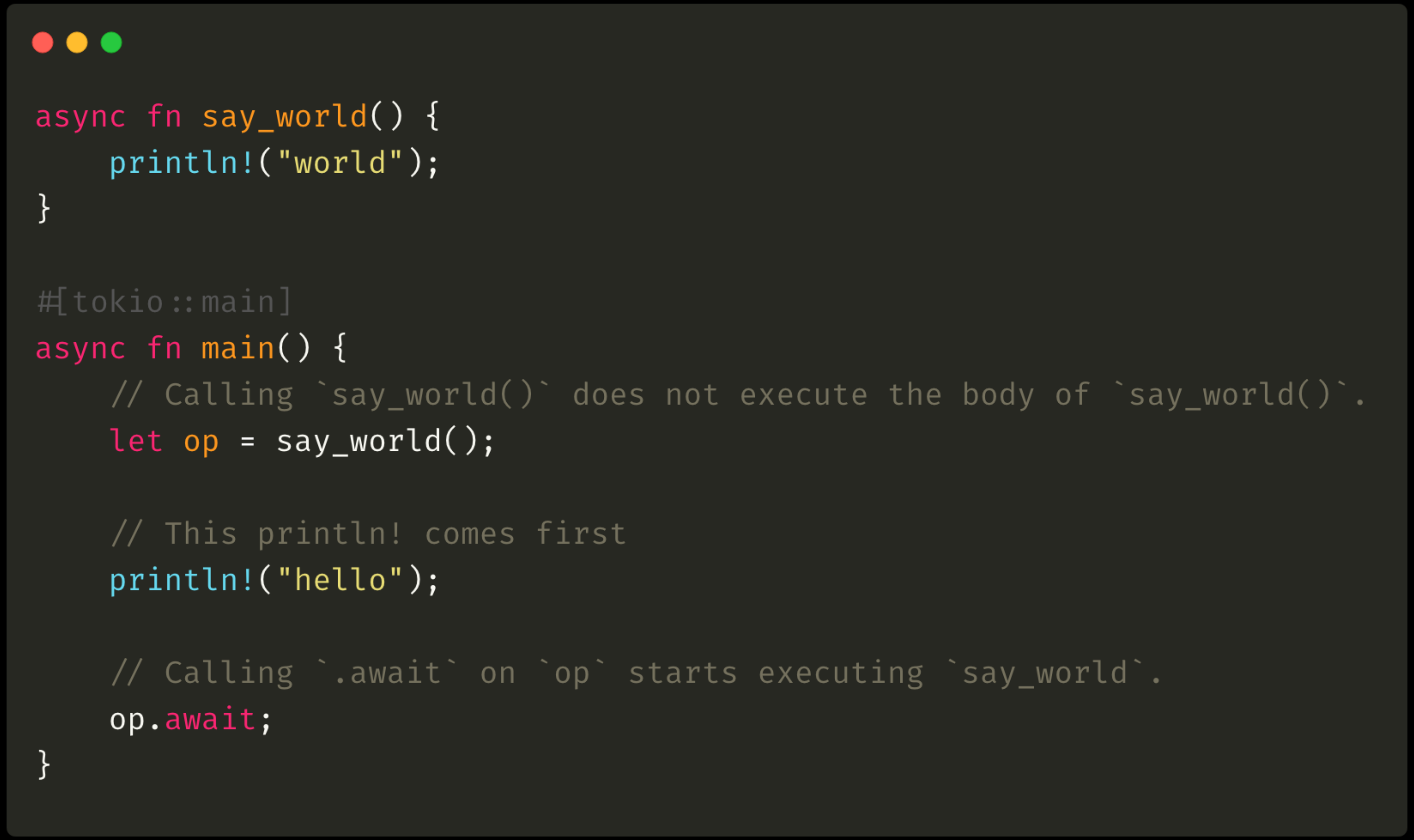


```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```

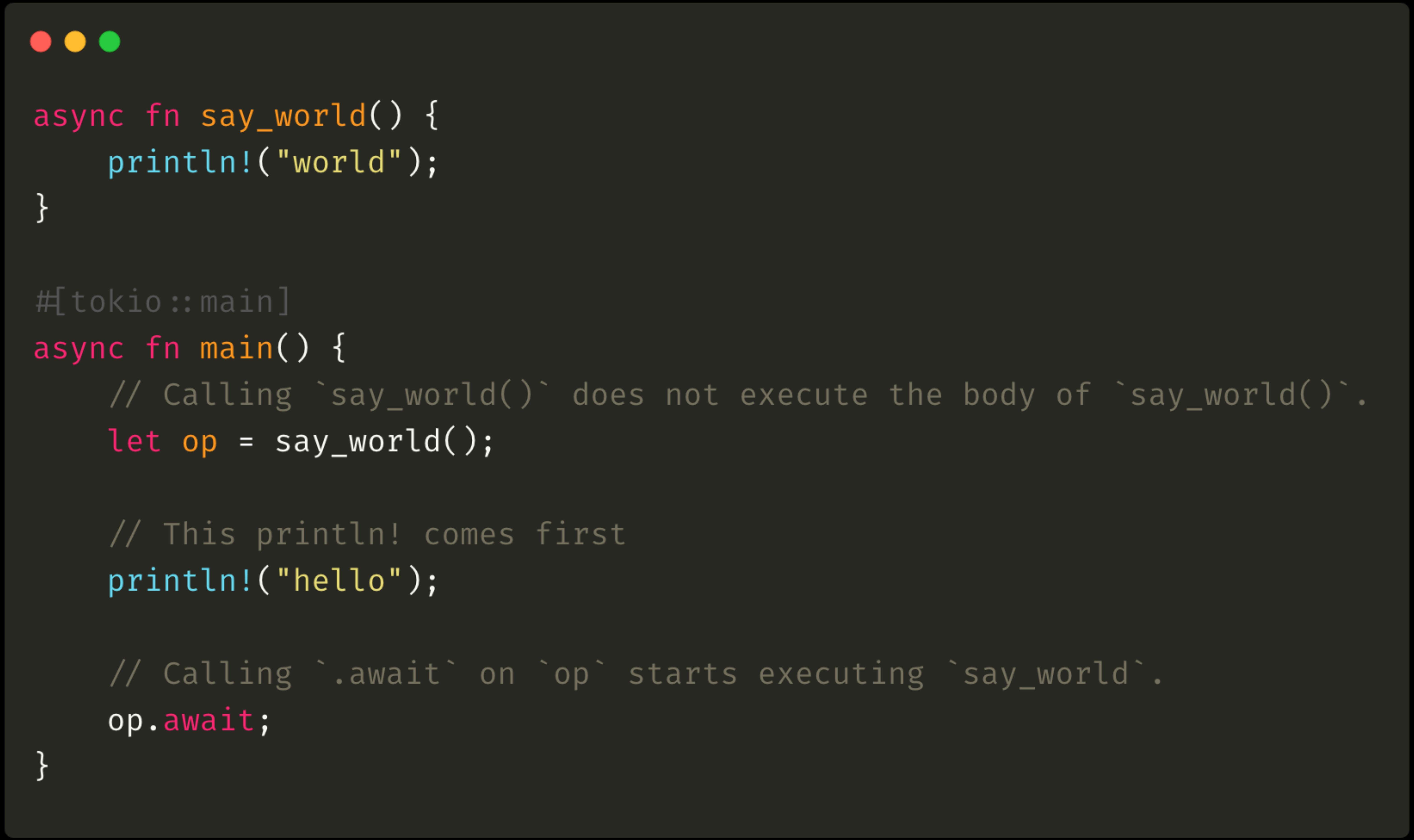


```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```



```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```

```
hello world
```

Enhance documentation

All crates
Click or press ‘S’ to search, ‘?’ for more options...
?
Struct Vec
Methods
append
as_mut_ptr
as_mut_slice
as_ptr
as_slice
capacity
clear
dedup
dedup_by
dedup_by_key
drain
drain_filter
extend_from_slice
from_raw_parts
1.0.0 [−][src]
[+] Show declaration
[−]
Run

Struct std::vec::Vec

A contiguous growable array type, written `Vec<T>` but pronounced 'vector'.

Examples

```
let mut vec = Vec::new();
vec.push(1);
vec.push(2);

assert_eq!(vec.len(), 2);
assert_eq!(vec[0], 1);

assert_eq!(vec.pop(), Some(2));
assert_eq!(vec.len(), 1);

vec[0] = 7;
assert_eq!(vec[0], 7);

vec.extend([1, 2, 3].iter().copied());
```

All crates
Click or press ‘S’ to search, ‘?’ for more options…
?
Struct Vec
Methods
append
as_mut_ptr
as_mut_slice
as_ptr
as_slice
capacity
clear
dedup
dedup_by
dedup_by_key
drain
drain_filter
extend_from_slice
from_raw_parts
1.0.0 [−][src]
[+] Show declaration
Run

Struct std::vec::Vec

[−] A contiguous growable array type, written `Vec<T>` but pronounced 'vector'.

Examples

```
let mut vec = Vec::new();
vec.push(1);
vec.push(2);

assert_eq!(vec.len(), 2);
assert_eq!(vec[0], 1);

assert_eq!(vec.pop(), Some(2));
assert_eq!(vec.len(), 1);

vec[0] = 7;
assert_eq!(vec[0], 7);

vec.extend([1, 2, 3].iter().copied());
```

What about dependencies?

Rust Playground is limited

… which complicates learning

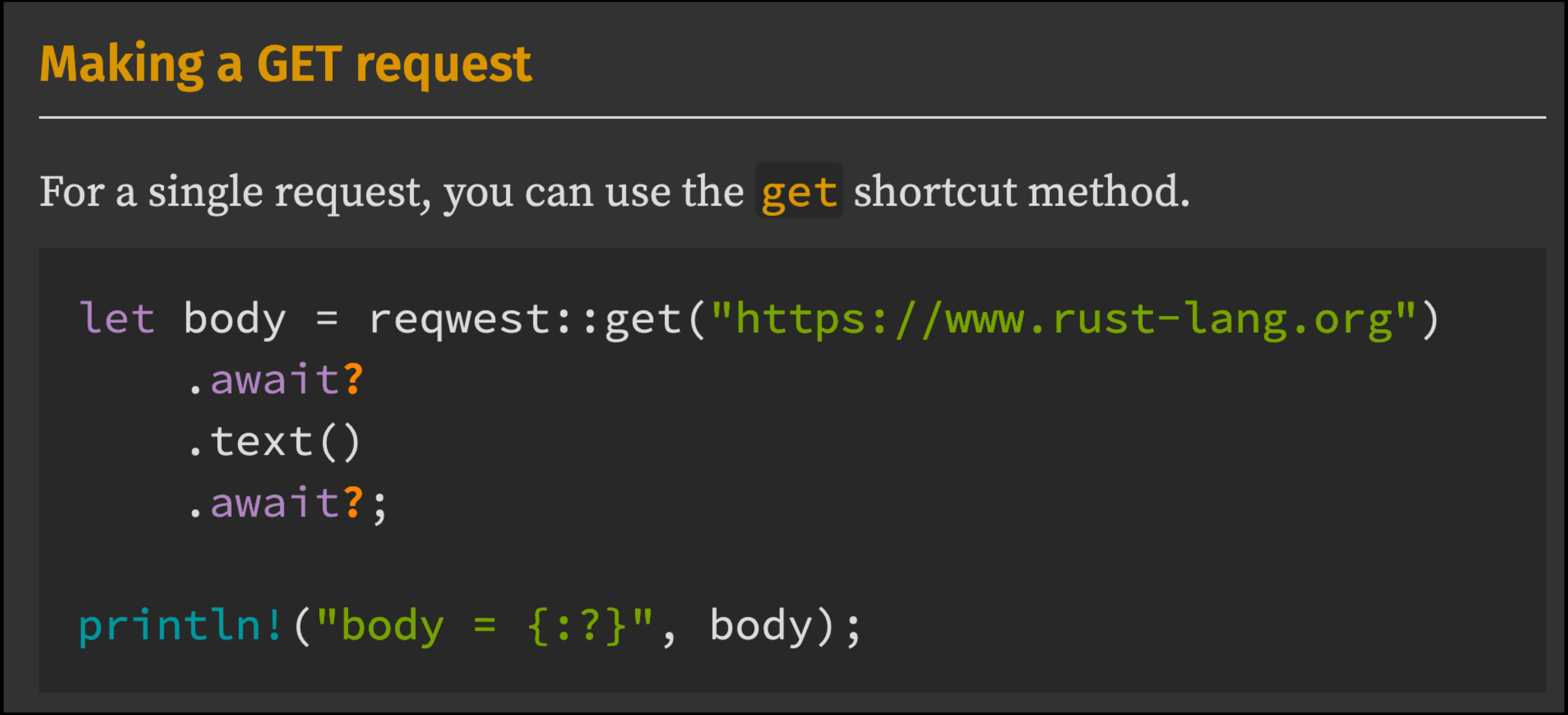
WebAssembly to save the day

So what about compatibility?

Mocks & stubs make it easy

Feature flags or auto-mocking

Visualize state

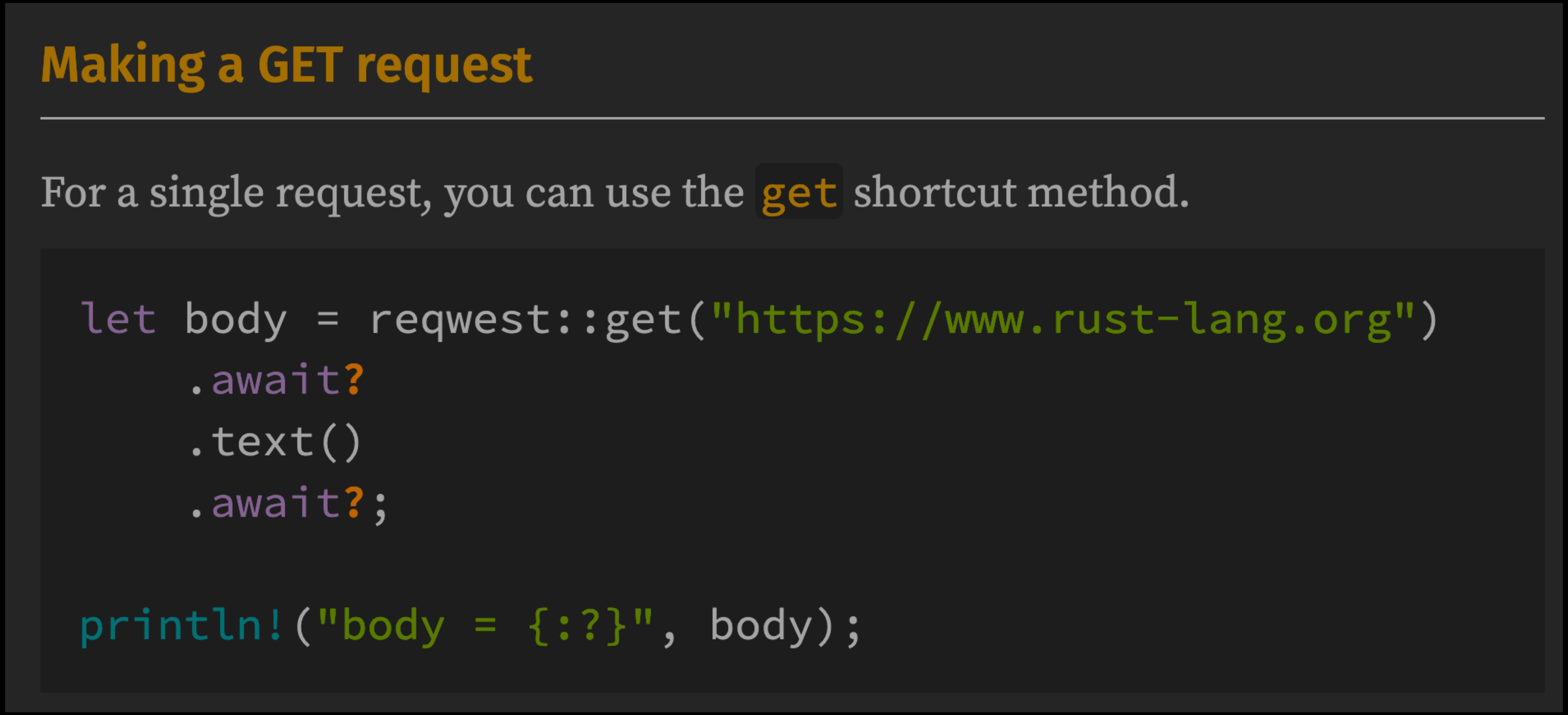


Making a GET request

For a single request, you can use the `get` shortcut method.

```
let body = reqwest::get("https://www.rust-lang.org")
    .await?
    .text()
    .await?;

println!("body = {:?}", body);
```



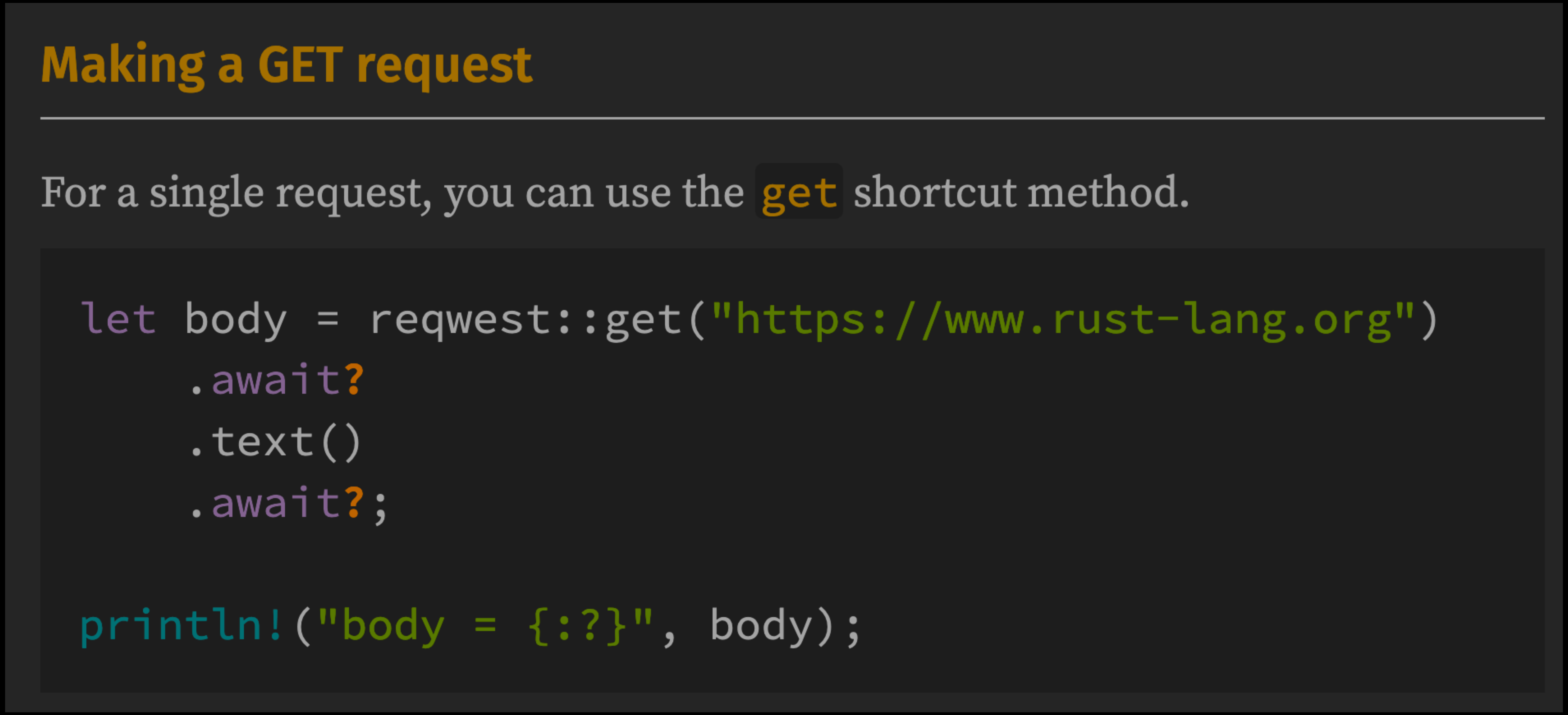
Making a GET request

For a single request, you can use the `get` shortcut method.

```rust
let body = reqwest::get("https://www.rust-lang.org")
    .await?
    .text()
    .await?;

println!("body = {:?}", body);
```

```
GET /
Host: www.rust-lang.org
User-Agent: reqwest
Accept: */*
```



Making a GET request

For a single request, you can use the `get` shortcut method.

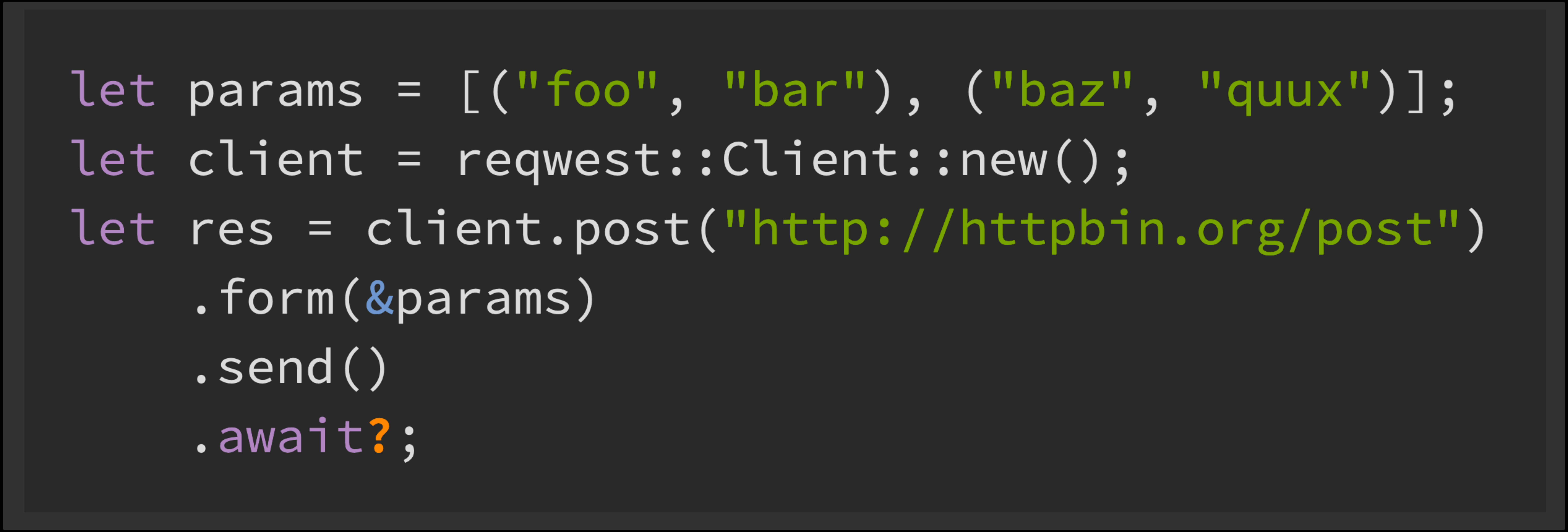
```rust
let body = reqwest::get("https://www.rust-lang.org")
    .await?
    .text()
    .await?;

println!("body = {:?}", body);
```

```
HTTP/1.1 200 OK
Server: Rocket
Content-Type: text/html
Content-Length: 65303
```

Highlight context

Code is data



```rust
let params = [("foo", "bar"), ("baz", "quux")];
let client = reqwest::Client::new();
let res = client.post("http://httpbin.org/post")
    .form(&params)
    .send()
    .await?;
```

```
let params = [("foo", "bar"), ("baz", "quux")];
let client = reqwest::Client::new();
let res = client.post("http://httpbin.org/post")
    .form(&params)  ← e.g.: foo=bar&baz=quux
    .send()
    .await?;
```

```
let params = [("foo", "bar"), ("baz", "quux")];
let client = reqwest::Client::new();
let res = client.post("http://httpbin.org/post")
    .form(&params)
    .send()
    .await?;
```

```
ExprMethodCall { method: “send” }
ExprMethodCall { method: “form” }
ExprMethodCall { method: “send” }
ExprAwait
```

```
match expr {
  ExprMethodCall { method_name } => { … }
}
```

Visualize execution

```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
      .filter(|&(i, x)| (i + x) % 3 == 0)
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
      .filter(|&(i, x)| (i + x) % 3 == 0)
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
(0..5).flat_map(|x| x * 100 .. x * 110)
```

```
x = 1, return: {100, 101, … 110 }
```

```
      .filter(|&(i, x)| (i + x) % 3 == 0))
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
       value = 1, return: (0, 100)                         ))
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
(0..5).flat_map(|x| x * 100 .. x * 110)
       .enumerate()
       .filter(|&(i, x)| (i + x) % 3 == 0)
        i = 1, x = 100, return: false           i, x));
```

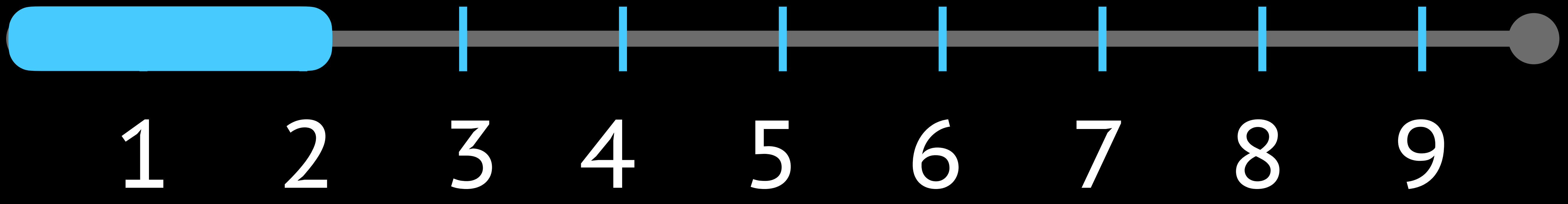
```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
```

```
value = 1, return: (0, 100)
```

```
                                           ))
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
(0..5).flat_map(|x| x * 100 .. x * 110)
    .enumerate()
                                           ))
    .for_each(|(i, x)| println!("{}:{}", i, x));
```

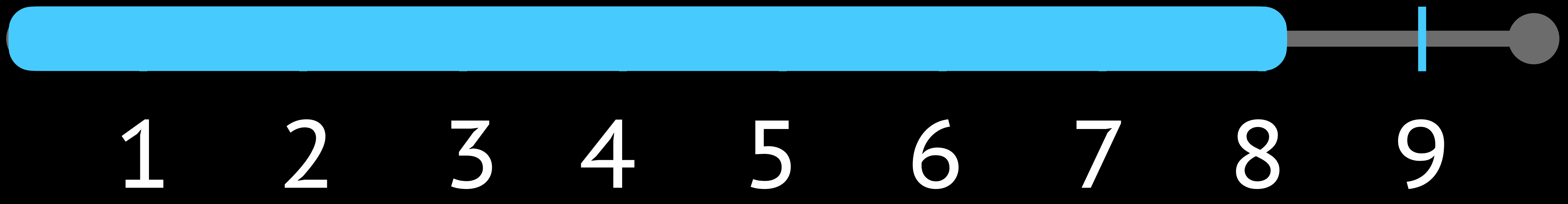
```
value = 1, return: (0, 100)
```



```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
```

```
i = 4, x = 104, “4:104”
```

```
      .for_each(|(i, x)| println!("{}:{}", i, x));
```



```
(0..5).flat_map(|x| x * 100 .. x * 110)
      .enumerate()
      .filter(|&(i, x)| (i + x) % 3 == 0)
      .for_each(|(i, x)| println!("{}:{}", i, x));
```

```
fn step1(&self) -> impl Iterator {
  (0..5).flat_map(|x| x * 100 .. x * 110)
}

fn step2(&self) {
  self.enumerate()
}
```

```
struct Snippet { <state> }

impl Generator for Snippet {
  fn step1(&self) -> impl Iterator {
    (0..5).flat_map(|x| x * 100 .. x * 110)
  }
  fn step2(&self) {
    self.enumerate()
  }
}
```

```
let snippet = Snippet::new().
snippet.step1();
// output the current state
snippet.step2();
// output the current state
```

How to implement it?

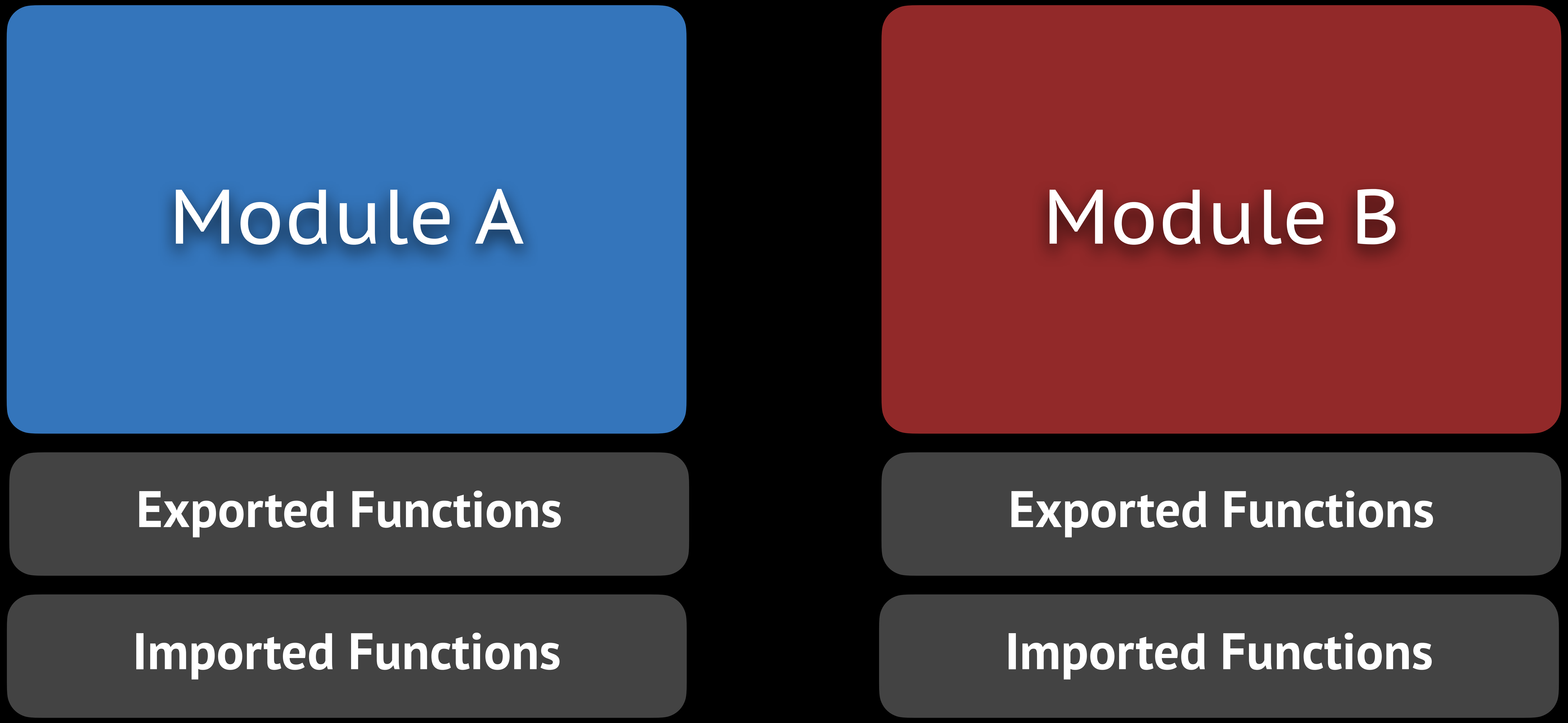
Infrastructure

Is it scalable?

Dependencies are hard

There’s no linking for WebAssembly

… or is there?



Module A

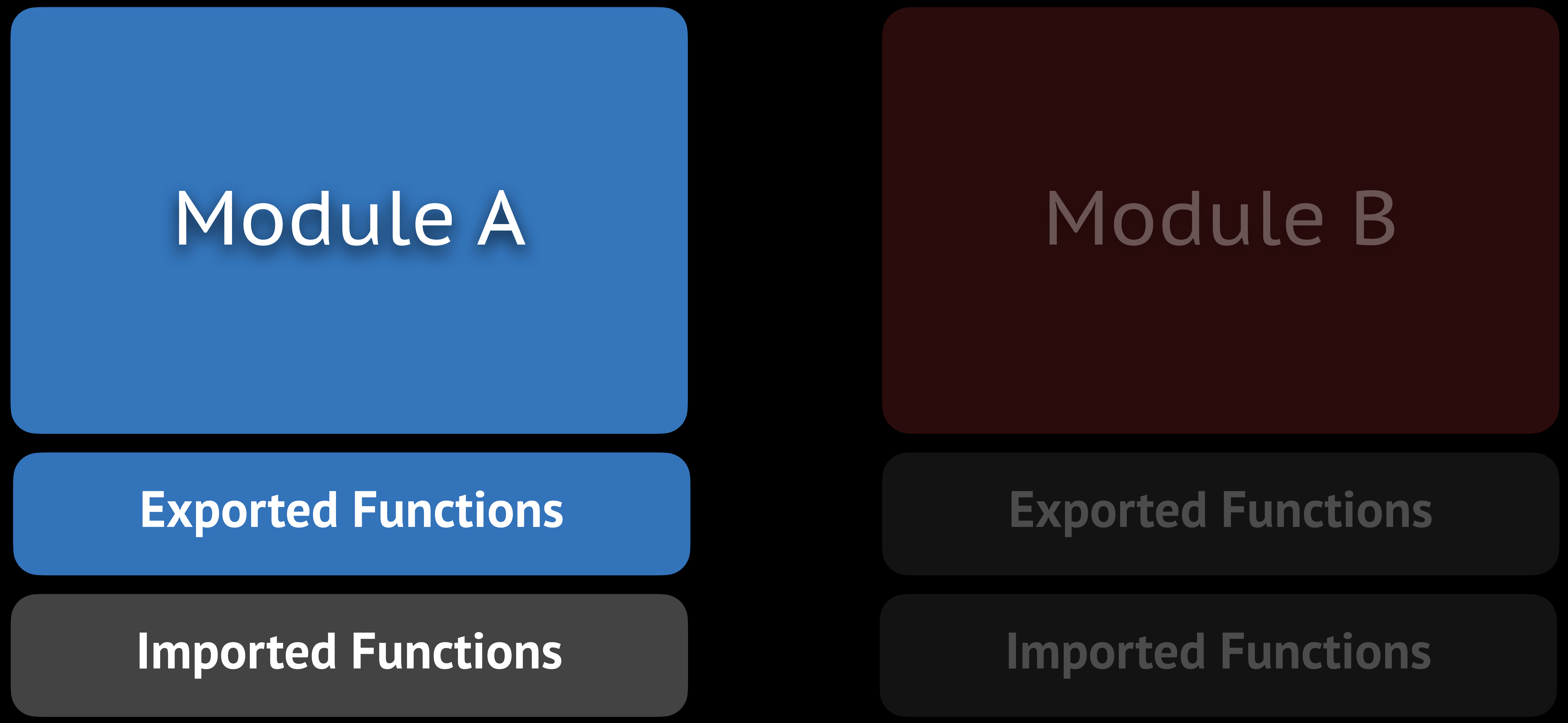
Exported Functions

Imported Functions

Module B

Exported Functions

Imported Functions



Module A

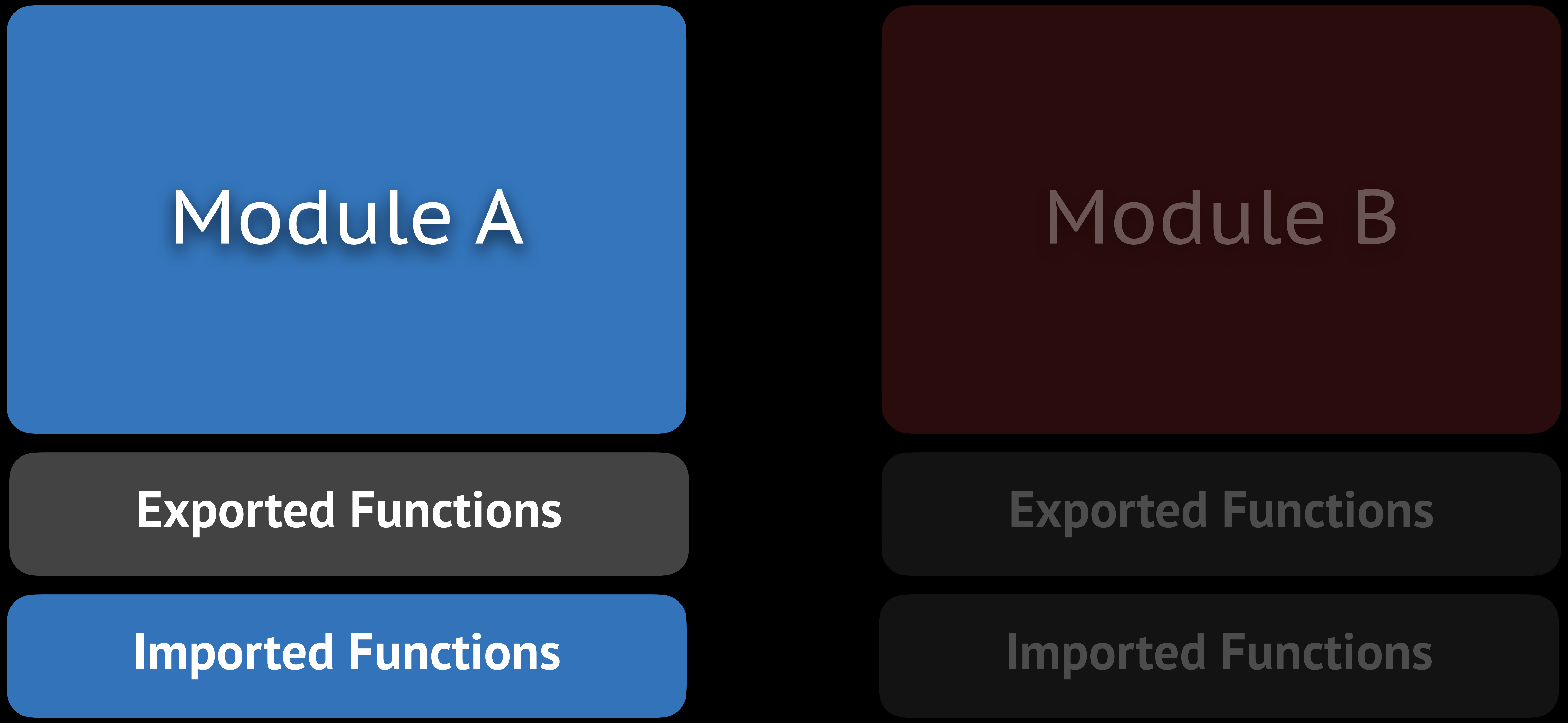
Exported Functions

Imported Functions

Module B

Exported Functions

Imported Functions



Module A

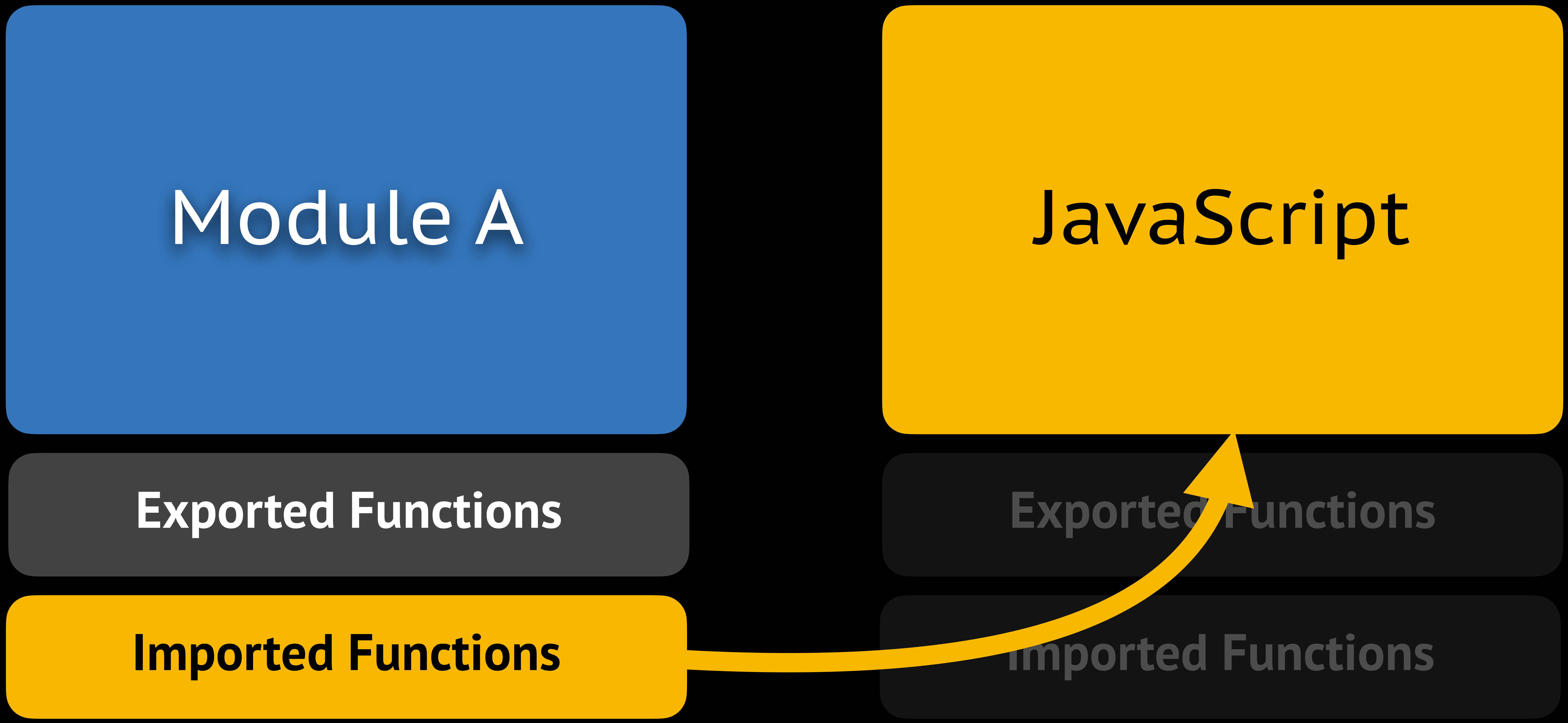
Exported Functions

Imported Functions

Module B

Exported Functions

Imported Functions



Module A

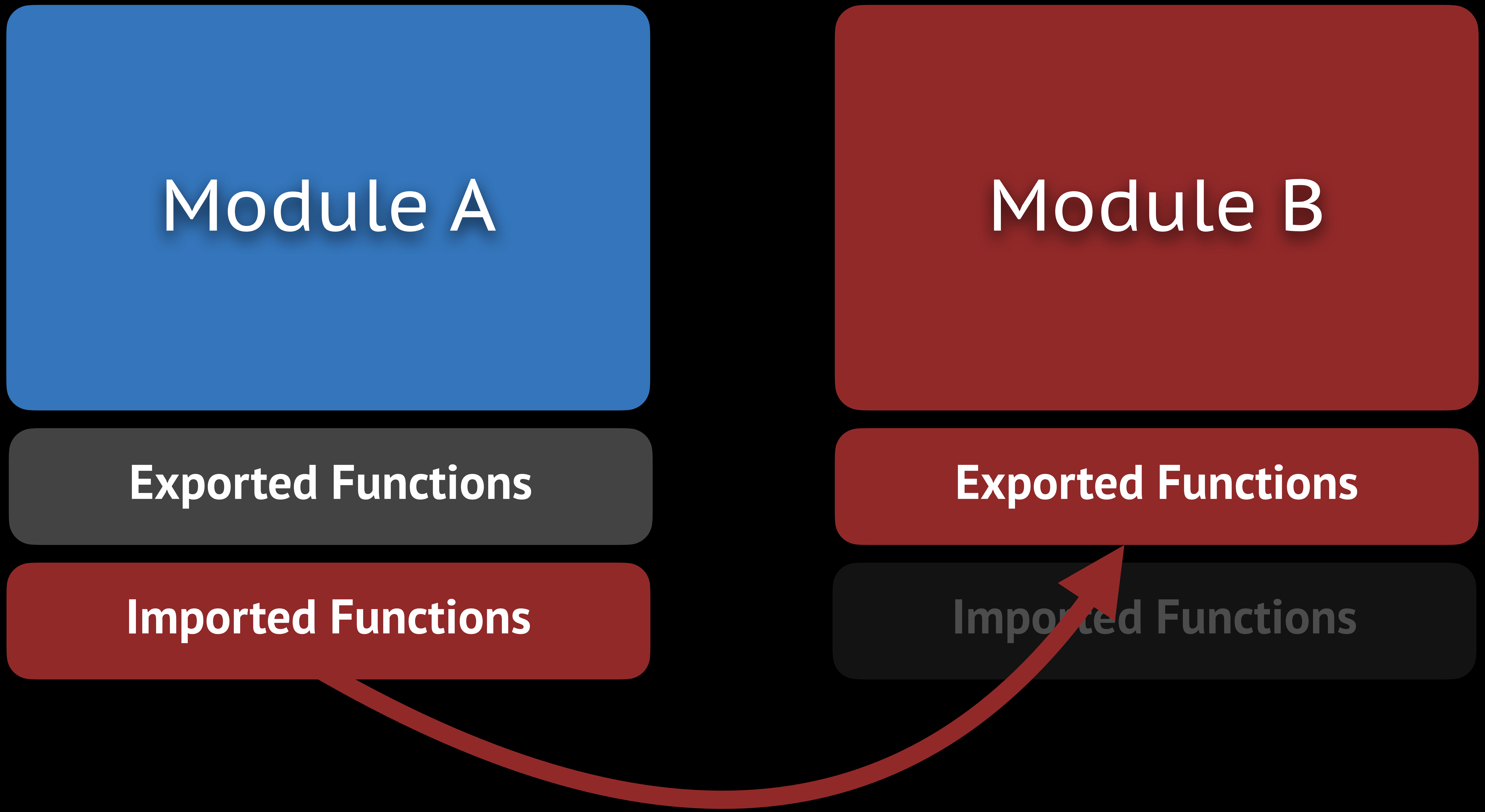
Exported Functions

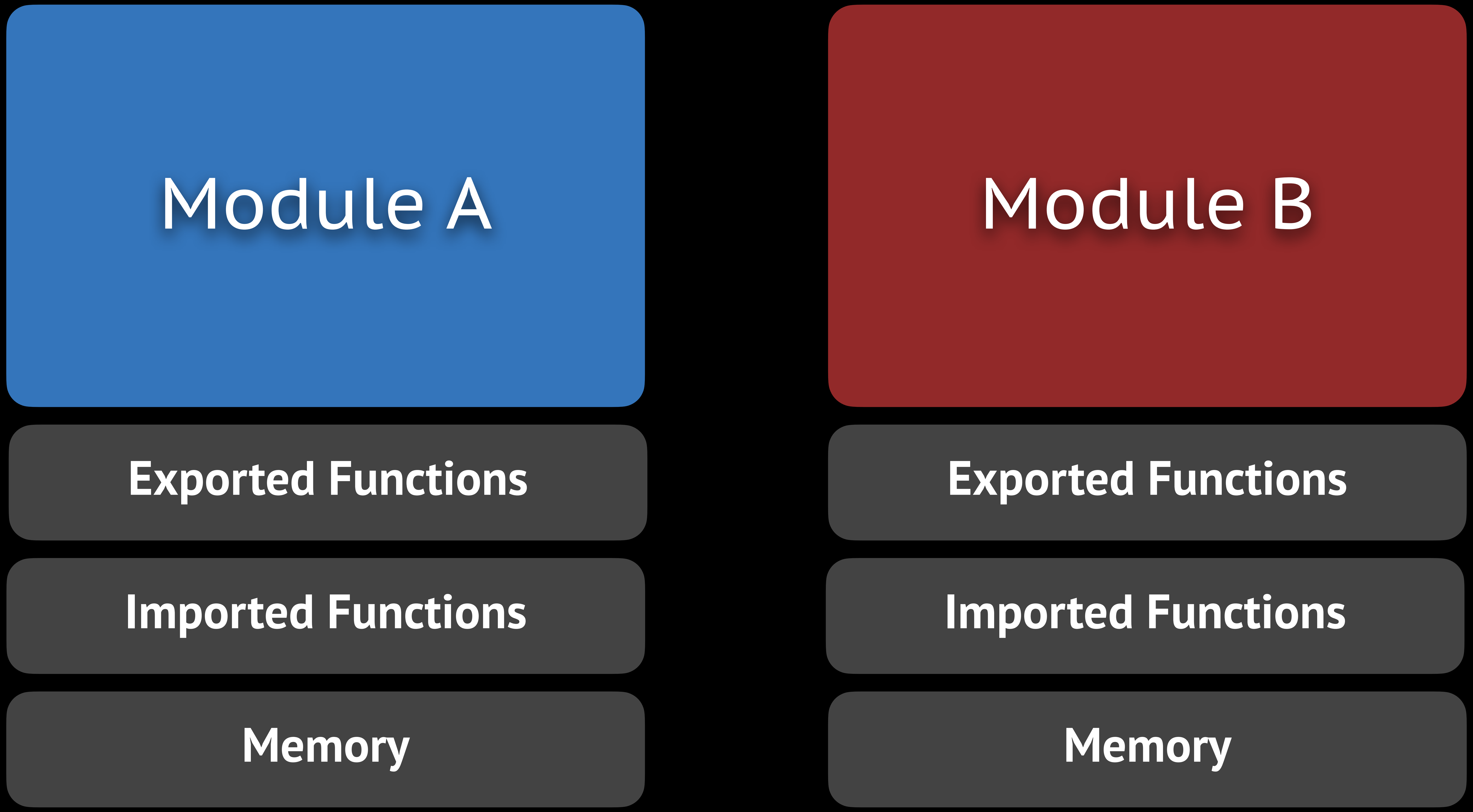
Imported Functions

JavaScript

Exported Functions

Imported Functions


Module A
Exported Functions
Imported Functions
Module B
Exported Functions
Imported Functions



Module A

Exported Functions

Imported Functions

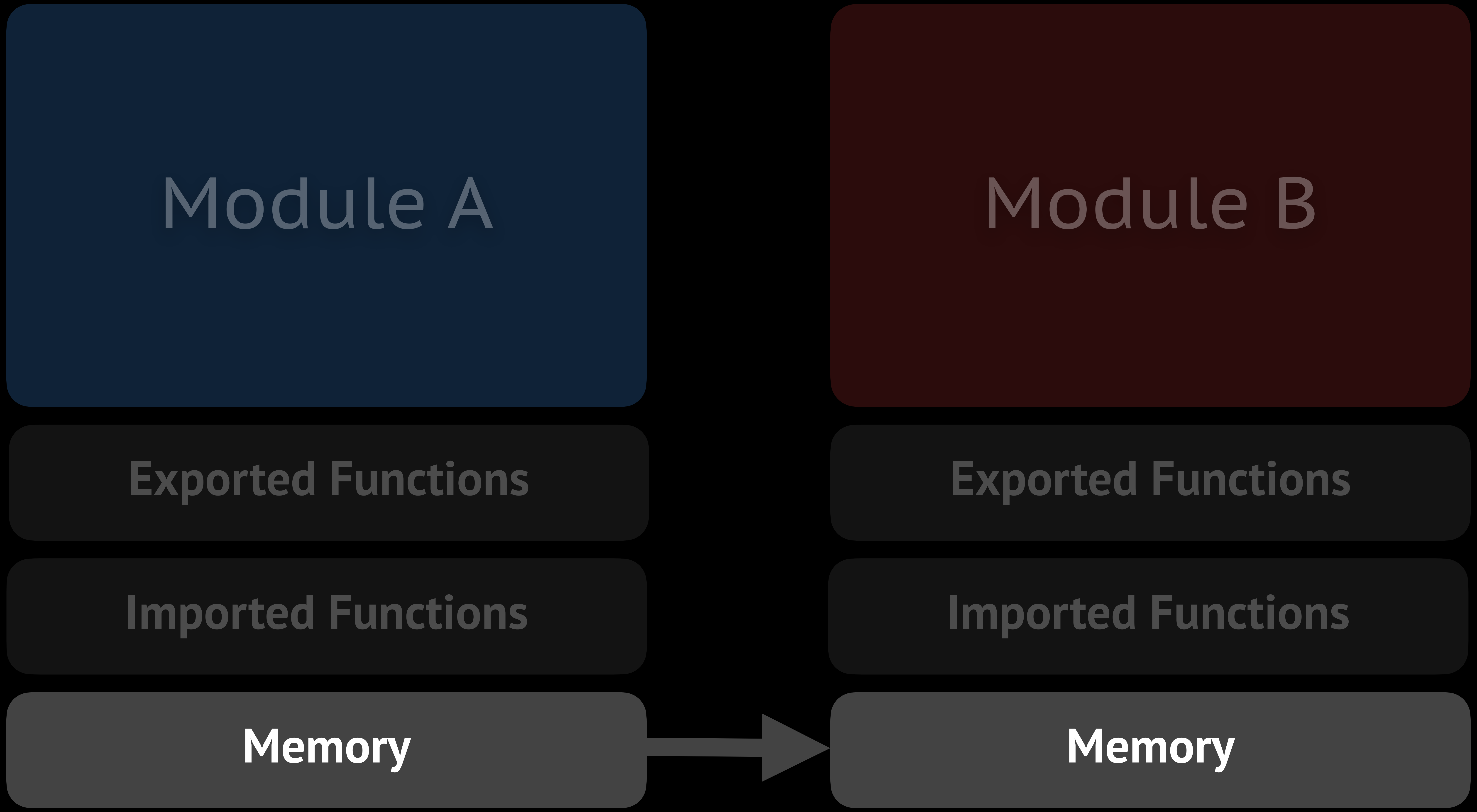
Memory

Module B

Exported Functions

Imported Functions

Memory



Module A

Exported Functions

Imported Functions

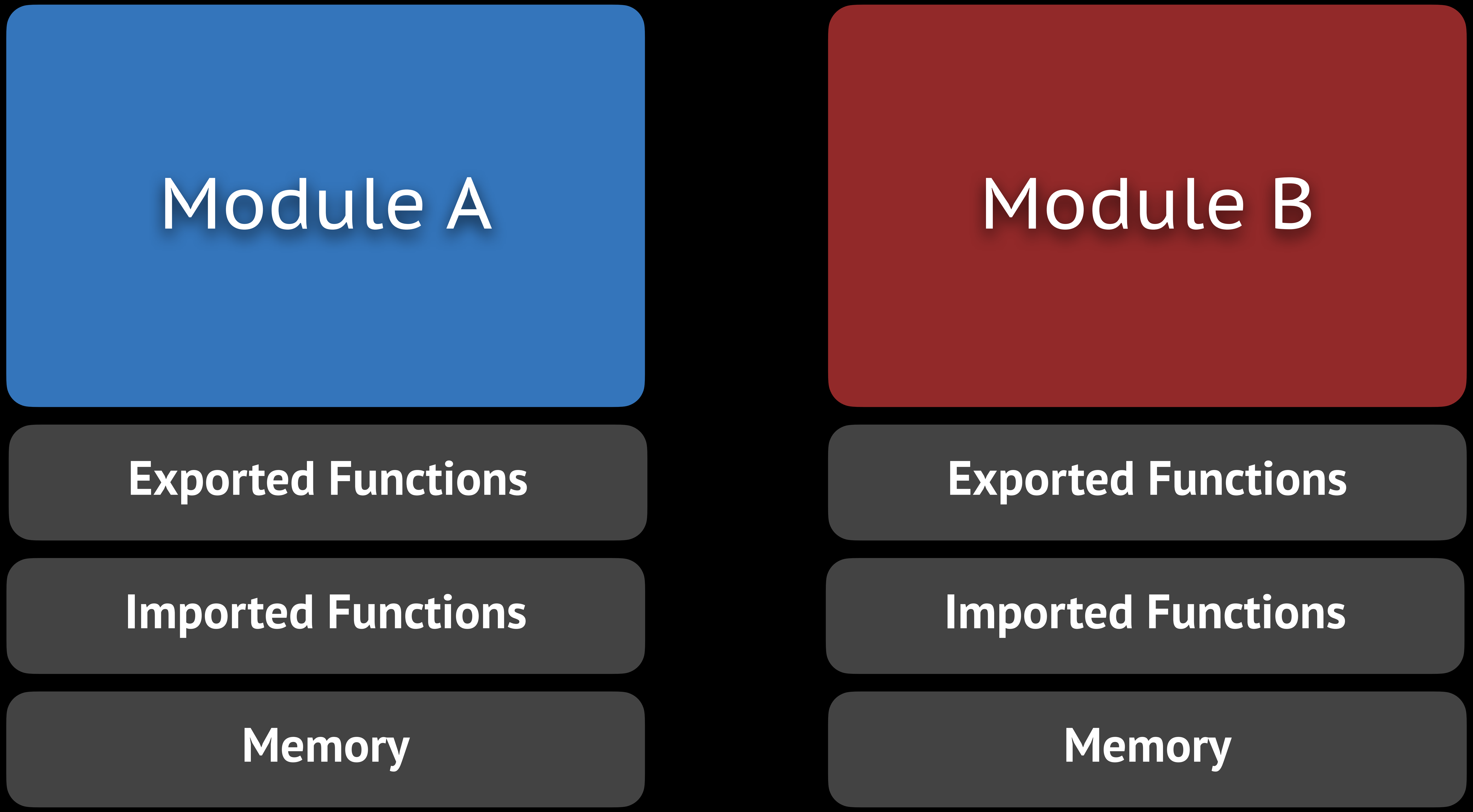
Memory

Module B

Exported Functions

Imported Functions

Memory



Module A

Exported Functions

Imported Functions

Memory

Module B

Exported Functions

Imported Functions

Memory

What’s next?

Make documentation interactive

Make it simple

Make it automatic

Join the development!

https://github.com/nbaksalyar/interactivedoc

Thank you!