

HTTP & Asynchrony in Dart

Jakub Fijałkowski
@Fiolek_
<https://codinginfinity.me>

HTTP in Dart

What is HTTP?

What is HTTP?

- A layer 7,
- Stateless,
- Connection-less,
- Textual,
- Request-response protocol,
- for transmitting hypermedia documents. :)

HTTP

- Verbs,
- Headers,
- Request/Response,
- Cookies,
- Status codes,
- Content.



Anatomy of HTTP request/response

GET / HTTP/1.1

Host: www.example.com

User-Agent: Mozilla/5.0

Accept:

text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8

Accept-Language: en-GB,en;q=0.5

Accept-Encoding: gzip, deflate, br

Connection: keep-alive

HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 155
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

```
<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Hello World, this is a very simple HTML document.</p>
  </body>
</html>
```


JSON

```
{  
  "firstName": "John",  
  "age": 27,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
  },  
  "phoneNumbers": [  
    {  
      "type": "home",  
    },  
    {  
      "type": "office",  
      "number": "646 555-4567"  
    }  
  ],  
  "spouse": null  
}
```

x-www-form-urlencoded

```
field1=value1&field2=value2
```

HTTP in Dart

We'll cover client-side only

httpbin.org^{0.9.2}

[Base URL: httpbin.org/]

A simple HTTP Request & Response Service.

Run locally: `$ docker run -p 80:80 kennethreitz/httpbin`

[the developer - Website](#)

[Send email to the developer](#)

Schemes

HTTP



HTTP Methods

Testing different HTTP verbs

Auth

Auth methods

dart:io's HttpClient

dart:io's HttpClient

- Built-in HTTP client,
- Quite bare,
- Rather awkward,
- Does not need external dependencies,
- Works in “plain” Dart & in Flutter (both web & native),
- Don't use it for anything more complex.

```
Future<void> simpleGet() async {
  final client = HttpClient();
  try {
    // Initiate a request - this opens the connection
    final request = await client.getUrl(anything);
    // Here, the underlying TCP connection is already opened but
    // the request has not been sent. We can configure it
    request.headers.add('We-Will', 'Rock You');

    // This line sends the request and reads (and buffers) the response
    final response = await request.close();

    // Now we have the response
    print('Status code: ${response.statusCode}');
    print('Reason phrase: ${response.reasonPhrase}');
    print('Content length: ${response.contentLength}');
    print('Headers:');
    response.headers.forEach((n, v) => print('\t$n = ${v.join('!')}'));

    // And we can read the body, but doing so is really cumbersome
    final data = await response.reduce((p, e) => p + e);
    final asString = String.fromCharCode(data);
    print('Body:');
    print(asString);
  } finally {
    // And we need to remember to close the client, otherwise
    // we might leak native resources
    // `force` is just for fun here :)
    client.close(force: true);
  }
}
```



```
Future<void> harderPost() async {
  final client = HttpClient();
  try {
    final request = await client.postUrl(anything);

    // We can set the cookies
    request.cookies.add(Cookie('Steve', 'walkswarilydownthestreet'));

    // And set the body, aka. content type
    request.headers.contentType = ContentType.text;
    request.headers.contentType = ContentType.json; // And we can update it

    // We don't need to send all at once
    request.write('{');

    // We can send chunks and flush in the process, as the data is buffered
    request.write('"another one": "bites the dust"');
    await request.flush();

    request.write('}');

    // But this will throw, as we've already sent (part of) the body :)
    // request.headers.contentType = ContentType.text;

    final response = await request.close();
    await debugResponse(response);
  } finally {
    client.close();
  }
}
```

```
Future<void> clientFailure() async {  
    final client = HttpClient();  
    try {  
        // If the server sends a response, we _always_ succeed  
        final request = await client.getUrl(Uri.parse('$statusCode/404'));  
        final response = await request.close();  
        await debugResponse(response);  
    } finally {  
        client.close();  
    }  
}
```

```
Future<void> networkFailure() async {  
    final client = HttpClient();  
    try {  
        // EXCEPT when the error is "ambient"  
        final request = await client.getUrl(invalidAddress);  
        final response = await request.close();  
        await debugResponse(response);  
    } finally {  
        client.close();  
    }  
}
```

```
Future<void> urlencoding() async {  
  final client = HttpClient();  
  try {  
    final request = await client.postUrl(anything);  
  
    request.headers.contentType = ContentType(  
      'application',  
      'x-www-form-urlencoded',  
    );  
    request.write(  
      'allwehear=${Uri.encodeComponent('radio goo goo')}&'  
      'radio=${Uri.encodeComponent('ga ga')}',  
    );  
  
    final response = await request.close();  
    await debugResponse(response);  
  } finally {  
    client.close();  
  }  
}
```

```
Future<void> postJson() async {
  final client = HttpClient();
  try {
    final request = await client.postUrl(anything);

    // We need to say that it's JSON
    request.headers.contentType = ContentType.json;

    // We need to serialize data to JSON manually
    request.write(json.encode({
      "I": "want to",
      "break": "free",
    }));

    final response = await request.close();

    // And deserialize it on our own, probably skipping all the
    // streaming capabilities :(
    final deserialized = json.decode(await readAll(response));
    print('Body:');
    printPrettyJson(deserialized, indent: 2);
  } finally {
    client.close();
  }
}
```

```
Future<void> files() async {
  const kBoundary = 'NOESCAPEFROMREALITY';
  final client = HttpClient();
  try {
    final request = await client.postUrl(anything);

    request.headers.contentType = ContentType(
      'multipart',
      'form-data',
      parameters: {'boundary': kBoundary},
    );
    request.writeAll(
      [
        '--$kBoundary',
        'Content-Disposition: form-data; name="test"; filename="test.json"',
        'Content-Type: application/json',
        '',
        '{"Is this the real life": "Is this just fantasy"}',
        '--$kBoundary--',
      ],
      '\r\n',
    );

    final response = await request.close();
    await debugResponse(response);
  } finally {
    client.close();
  }
}
```

http package

http.Client

- Wrapper over (possibly) `dart:io`'s `HttpClient`,
- But can use different mechanisms (it is pluggable),
- Adds a little bit of functionality to `HttpClient`,
- Makes common actions easier,
- But still lacks *many* features,
- But adding them is much easier than to a plain `HttpClient`,
- A middle ground between bare `HttpClient` & `dio`, especially if you don't need `dio`'s complexity.


```
Future<void> simpleGet() async {
    // Do the whole request _in a single call_
    final response = await http.get(
        anything,
        headers: {
            'Tonight': 'we\'re summoned for a divine cause',
        },
    );

    // We have all the common fields available
    print('Status code: ${response.statusCode}');
    print('Reason phrase: ${response.reasonPhrase}');
    print('Content length: ${response.contentLength}');
    print('Headers:');

    // THIS IS WRONG - headers don't need to be unique but this model
    // assumes that they are. They are concatenated underneath with ', '.
    // This is allowed by the HTTP spec but the client _forces_
    // this desing, whereas HTTP only _allows_ it.
    response.headers.forEach((n, v) => print('\t$n = $v'));

    // And body is easily available :)
    print('Body:');
    print(response.body);
}
```

```
Future<void> harderPost() async {
  final response = await http.post(
    anything,
    headers: {
      // No proper cookie handling :(
      'Set-Cookie': 'hidingfromthelight=sacrificingnothing',

      // We can't set it when we set `body` to a Map
      // 'Content-Type': 'application/json',
    },
    body: <String, String>{
      // We can use maps and leave serialization to the library :D
      'Are you on the square?': 'Are you on the level?',
    },
  );

  await debugResponse(response);
  // But there's a catch...
}
```

```
Future<void> harderPostCorrect() async {  
  // `http` does not know about JSON :)  
  final response = await http.post(  
    anything,  
    headers: {  
      // No proper cookie handling :(  
      'Set-Cookie': 'justwannabe=wannabewitchyou',  
  
      'Content-Type': 'application/json',  
    },  
    body: '{ "It keeps on giving me chills": "But I know now" }',  
  );  
  
  await debugResponse(response);  
}
```

```
Future<void> clientFailure() async {  
    // Same behavior as `HttpClient`  
    final response = await http.get(Uri.parse('$statusCode/404'));  
    await debugResponse(response);  
}  
  
Future<void> networkFailure() async {  
    // Same here  
    final response = await http.get(invalidAddress);  
    await debugResponse(response);  
}
```

```
Future<void> urlencoding() async {  
  // Yep, it's that simple  
  final response = await http.post(  
    anything,  
    body: {  
      'Youvebeenplayinaround': 'with magic that is black',  
      'But all the powerful magical':  
        'mysteries never give a single thing back',  
    },  
  );  
  await debugResponse(response);  
}
```

```
Future<void> postJson() async {
  final response = await http.post(
    anything,
    headers: {
      // We need to specify that it's JSON
      'Content-Type': 'application/json; charset=utf-8',
    },
    // And serialize on our own
    body: json.encode({
      'Can you hear the rumble?': 'Can you hear the rumble that\'s calling?',
    }),
  );

  // And deserialize on our own, again - with no streaming
  final deserialized = json.decode(response.body);
  print('Body:');
  printPrettyJson(deserialized, indent: 2);
}
```

```
Future<void> files() async {
  final request = http.MultipartRequest('POST', anything)
    ..files.add(
      http.MultipartFile.fromString(
        'test',
        '{"You\'ll soon be hearing the chime": "Close to midnight"}',
        contentType: MediaType('application', 'json'),
      ),
    );

  // `response` here is a StreamedResponse - meaning that we don't
  // buffer (i.e. copy fully to memory) the output.
  final response = await request.send();

  await debugResponse(response);
}
```

dio

dio

- **The** HTTP client for Dart (& Flutter),
- Does not implement HTTP itself - it adapts existing implementations (so possibly `HttpClient`),
- The client is much easier to use,
- And supports many features not directly supported by `HttpClient` or `http.Client`,
- But adds much complexity,
- That is sometimes worth it,
- Especially if you need extensibility (and you probably will at some point).

```
Future<void> simpleGet() async {
  // Do the whole request _in a single call_
  final response = await Dio().getUri(
    anything,
    options: Options(
      headers: {
        'Mmmmmhhh-mhmm': 'yeah yeah',
      },
    ),
  );

  // We have all the common fields available
  print('Status code: ${response.statusCode}');
  // But ofc nothing can be perfect :)
  print('Reason phrase: ${response.statusMessage}');
  // But at least nothing is "promoted" artificially
  print('Content length: ${response.headers['Content-Length']}');
  print('Headers:');

  // Proper header handling
  response.headers.forEach((n, v) => print('\t$n = ${v.join(',')}'));

  // And body is easily available, with a catch :)
  print('Body:');
  // `data` is a `dynamic` and depends on options passed to the request
  print(response.data.toString());
}
```

```
Future<void> harderPost() async {
  final response = await Dio().postUri(
    anything,
    options: Options(
      headers: {
        // No proper cookie handling in the base
        // But there is dio_cookie_manager just for that :)
        'Set-Cookie': 'time=alongassfuckingtimeago',
      },
      // We can set the content type, but Dio is able to manage it itself
      // However, it will listen to us and not override it!
      contentType: Headers.jsonContentType,
    ),
    data: <String, String>{
      // We can use maps and leave serialization to the library :D
      'Dio can you hear me': 'I am lost and so alone',
    },
  );

  // Also, it is worth noting that the response is _also_ deserialized
  // According to content-type (if supported - JSON is :) )
  debugResponse(response);
}
```

```
Future<void> clientFailure() async {  
  // FINALLY! Consistent approach to error handling!  
  final response = await Dio().get('$statusCode/404');  
  debugResponse(response);  
}  
  
Future<void> networkFailure() async {  
  // :D  
  final response = await Dio().getUri(invalidAddress);  
  debugResponse(response);  
}
```

```
Future<void> urlencoding() async {
  // Just like that - specify the content type, Dio will manage
  // it for you
  final response = await Dio().postUri(
    anything,
    options: Options(contentType: Headers.formUrlEncodedContentType),
    data: {
      'Youwatchtheirfaces':
        'You\'ll see the traces of the things they want to be',
    },
  );
  debugResponse(response);
}
```

```
Future<void> postJson() async {
  // We can specify the `data` type in result
  // `Map<String, dynamic>` will be interpreted as JSON response
  final response = await Dio().postUri<Map<String, dynamic>>(
    anything,
    data: {
      'We\'ll know': 'for the first time',
      'Ifwere': 'evil or divine',
    },
  );

  debugResponse(response);
}
```

```
Future<void> files() async {  
  // Easily add files, even easier than in `http` (which was nice)  
  final data = FormData()  
    ..files.add(  
      MapEntry(  
        '1',  
        MultipartFile.fromString('No sign of the morning coming'),  
      ),  
    )  
    ..files.add(  
      MapEntry(  
        '2',  
        MultipartFile.fromString('You\'ve been left on your own'),  
      ),  
    );  
  final response = await Dio().postUri(anything, data: data);  
  debugResponse(response);  
}
```

```
Future<void> advanced2() async {  
  // Dio supports (easily) HTTP2 if available  
  final dio = Dio()..httpClientAdapter = Http2Adapter(ConnectionManager());  
  try {  
    final response = await dio.getUri(anything);  
    debugResponse(response);  
  } finally {  
    // But it's better to manage Dio lifecycle yourself  
    // As HTTP2 connections are quite long-living  
    dio.close(force: true);  
  }  
}
```



```
Future<void> advanced3() async {
  // Dio also supports `interceptor`, which allow you to do
  // things with request/response without being too explicit
  // everywhere. Like, logging :)
  final dio = Dio()..interceptors.add(LogInterceptor());
  try {
    final response = await dio.getUri(anything);
    print('The text above is printed directly by Dio');
    print('');
    debugResponse(response);
  } finally {
    dio.close(force: true);
  }
}
```

Summary

1. HTTP,
2. `dart:io HttpClient`,
3. `http.Client`,
4. `dio`



Asynchrony

Async/Concurrent vs parallel

Single thread of execution

Single thread of execution
≠
no asynchronous execution



It's just
“no one will interrupt my code”

i.e. DartVM does not support preemption



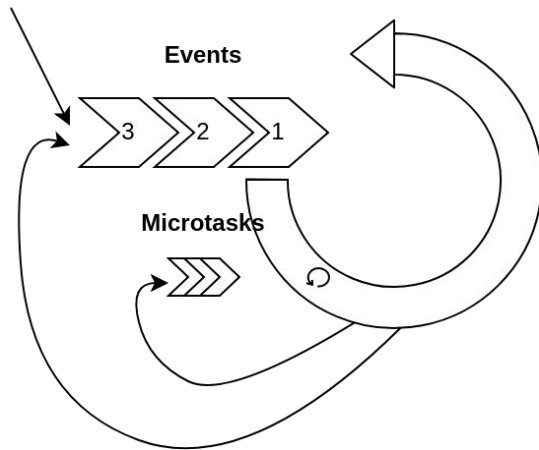
The same as in JavaScript

Who would have thought that JS is one of Dart's compilation targets. ;)

The event loop

I prefer calling it *the message loop*.

The event loop



The loop

```
var events = Queue<Function>();
var microtaskQueue = Queue<Function>();
events.add(main);

while (!events.isEmpty) {
  while (!microtaskQueue.isEmpty) {
    microtaskQueue.removeFirst().call();
  }
  events.removeFirst().call();
}
```

Takeaways

1. A single thread of execution does not mean no asynchrony,
2. It's just “no interruptions”,
3. It greatly simplifies thinking about the code - no one will change data behind your back,
4. But because of “no interruptions”, one can block the whole program with a single loop.
5. This programming model is not new, browsers work like that since '95.

How to model asynchrony?



With futures!



aka promises

What is a Future

- It's not parallel execution - it's just that we don't have the value yet,
- It's a *promise* of a value that will eventually be available.

Futures - async & await

```
Future<void> myFunc() async {  
    print("\tBefore delay");  
    await Future.delayed(const Duration(milliseconds: 500));  
    print("\tAfter delay");  
}  
  
Future<void> main() async {  
    print("Before call");  
    var prom = myFunc();  
    print("After call");  
    await prom;  
    print("After await");  
}
```

Futures = Promises

```
async function myFunc() {  
  console.log("\tBefore delay");  
  await new Promise(resolved => setTimeout(resolved, 500));  
  console.log("\tAfter delay");  
}  
  
async function main() {  
  console.log("Before call");  
  var prom = myFunc();  
  console.log("After call");  
  await prom;  
  console.log("After await");  
}
```

Future desugaring

```
Future main() async {  
  try {  
    var file = File("sample.txt");  
    var content = await file.readAsString();  
  
    print("File: ${content}");  
  } catch (e) {  
    print("ERROR: ${e}");  
  } finally {  
    print("All done");  
  }  
}
```

```
Future main() {  
  
  var file = File("sample.txt");  
  return file.readAsString().then((content) =>  
  
    print("File: ${content}"))  
    .catchError((e) =>  
      print("ERROR: ${e}"))  
    .whenComplete(() =>  
      print("All done"));  
}
```

Futures, events, microtasks - rules

1. The function that you pass into Future's **then()** method executes immediately when the Future completes. (The function isn't enqueued, it's just called.)
2. If a Future is *already complete* before **then()** is invoked on it, then a task is added to the *microtask queue*, and *that* task executes the function passed into **then()**.
3. The **Future()** and **Future.delayed()** constructors don't complete immediately; they add an item to the event queue.
4. The **Future.value()** constructor completes in a microtask, similar to #2.
5. The **Future.sync()** constructor executes its function argument immediately and (unless that function returns a Future) completes in a microtask, similar to #2.

<https://webdev.dartlang.org/articles/performance/event-loop>



What if we have multiple values?

Streams

Stream

- Models a *series*, possibly infinite, of values,
- Values can arrive at any moment in time, in no particular order,
- Future is more or less the same as `Stream` that reports one value and gets closed.

Streams usage

```
import "dart:io";
import "dart:convert";

Future main() async {
  var lines = File("sample.txt")
    .openRead()
    .transform(Utf8Decoder())
    .transform(new LineSplitter());
  await for (var l in lines) {
    print("Line: ${l}");
  }
  print("Done");
}
```


Streams desugaring

```
Stream<int> gen() async* {  
  
    yield 1;  
    await Future.delayed(  
        Duration(seconds: 1));  
    yield 2;  
}  
Future main() async {  
  
    await for (var i in gen()) {  
        print("Data: ${i}");  
    }  
    print("After");  
}
```

```
import "dart:async";  
Stream<int> gen() {  
    var ctrl = StreamController<int>();  
    new Future(() =>  
        ctrl.add(1))  
        .then((_) => Future.delayed(  
            Duration(seconds: 1)))  
        .then((_) { ctrl.add(2); ctrl.close(); });  
    return ctrl.stream;  
}  
Future main() {  
    var c = Completer();  
    gen().listen(  
        (i) => print("Data: ${i}"),  
        onDone: () => c.complete());  
    return c.future.then((_) => print("After"));  
}
```

Takeaways

1. Futures are like Promises in other languages (JS),
2. Future represents “data that will be available some time in the future”,
3. Streams are used to model lists (well... iterables) of values that will be generated asynchronously,
4. In other words

	Sync	Async
Single value	T	Future<T>
Multiple values	Iterable<T>	Stream<T>

Aspects of asynchrony

Enter the Zone

“A zone represents the asynchronous dynamic extent of a call. It is the computation that is performed as part of a call and, transitively, the asynchronous callbacks that have been registered by that code.”

<https://www.dartlang.org/articles/libraries/zones>

Zone

- Might define async error boundary,
- Controls some of the aspects of execution (i.e. callbacks, timers, microtasks),
- Provides storage of zone-local values that follow the async execution flow.

Error boundary

```
import "dart:async";

void main() {
  var fut = Future.value(100);
  var res = runZoned(() {
    return fut.then((_) => throw "inside zone");
  }, onError: (e, s) => print("Zoned: ${e}"));
  res.then(print)
    .catchError((e) => print("Outside: ${e}"));
}
```

Controlling execution

```
import "dart:async";

void func() => print("In Func");

void main() {
  runZoned(() async {
    func();
    await new Future(() => print("In Future"));
    print("After future");
  }, zoneSpecification: ZoneSpecification(
    print: (self, parent, zone, s) => parent.print(zone, "Zoned: ${s}"),
    run: <R>(self, parent, zone, f) {
      parent.print(zone, "Zone call: ${f}");
      return parent.run(zone, f);
    }));
}
```

Zone-local storage

```
void myWork(String otherData) {  
    print(otherData);  
}  
  
void processArg(String arg) {  
    uncontrolled("Data gen");  
}  
  
void uncontrolled(String otherData) => myWork(otherData);  
  
void main() => ['A', 'B'].forEach(processArg);
```


Zone-local storage

```
import "dart:async";

void myWork(String otherData) {
  print("${Zone.current[#arg]} -> ${otherData}");
}

void processArg(String arg) {
  runZoned(() => uncontrolled("Data gen"),
    zoneValues: { #arg: arg });
}

void uncontrolled(String otherData) => myWork(otherData);

void main() => ['A', 'B'].forEach(processArg);
```

Takeaways

1. Zones allow to control async execution from the outside,
2. They define async boundaries,
3. Can be used as an ambient data storage,
4. They are prevalent in async code but most of the time you won't see them.

Parallel execution



Enter isolates



Isolate \approx thread



Isolate \approx process



You're always running in an
isolate

Isolate has

- Separate event loop,
- Separate(-ish) code,
- Separate address space.

Isolate limits

- You can't share data,
- You must use a static or top-level function as entrypoint, i.e. must not share state through closure,
- You can think of an isolate as a separate process (when comparing to Unix threading model).

Example

```
import "dart:isolate";

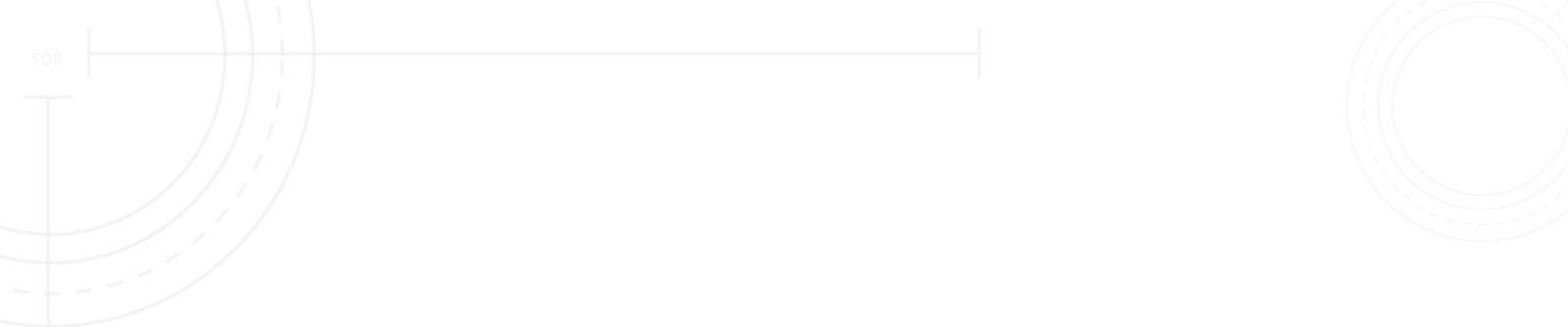
int value = 0;

void cpuIntensiveWork(int delay) {
  final sw = Stopwatch()..start();
  while (sw.elapsedMilliseconds < delay * 500);
}

void entryPoint(int msg) {
  print("[${msg}] Running in ${Isolate.current.hashCode}");
  while (value < 5) {
    print("[${msg}]: ${value++}");
    cpuIntensiveWork(msg);
  }
}

void main() {
  [1, 2].forEach((i) => Isolate.spawn(entryPoint, i));
  entryPoint(3);
}
```

Communication with isolates



Always two there are; no more, no less.
A SendPort and a ReceivePort.

This is somewhat a lie but I couldn't resist. ;)

Message-based communication

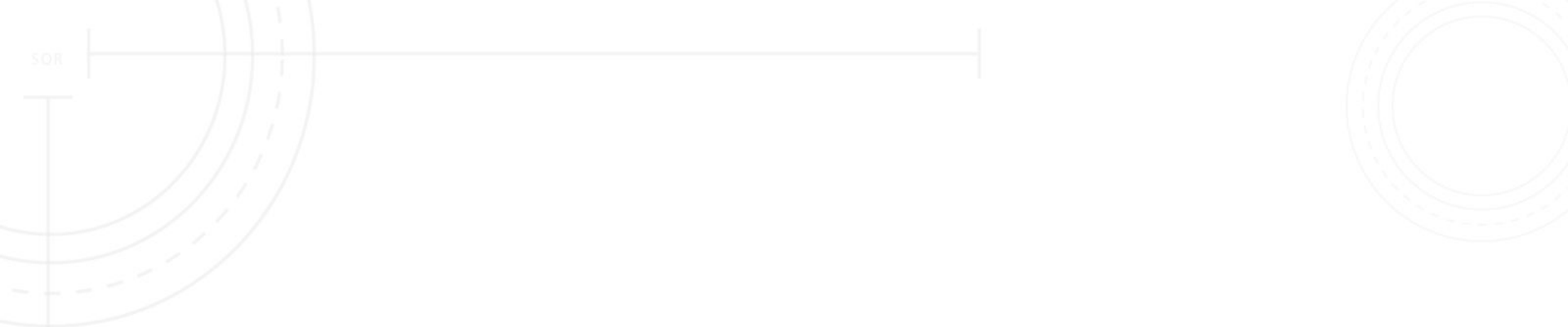
The only way to communicate with isolate is through ports and messages. Isolate can send a message to a `SendPort` and other isolate can receive message from the corresponding `ReceivePort`.

A `SendPort` always corresponds to a single `ReceivePort`. There can be multiple `SendPorts` that point to a single `ReceivePort` but we can treat them as copies of the same thing.

Ports marshal data

SendPort accepts:

- Primitive types + null,
- Lists of primitive types,
- Maps of primitive types,
- SendPorts and Capabilities,
- Objects IF isolates are in the same process and VM supports this (i.e. DartVM-only) but the objects are copied nevertheless.



This means you can't share references,
you must copy underlying data

Isolates & ports

```
import "dart:isolate";

void isolateEntry(SendPort controlPort) async {
  final dataPort = new ReceivePort();
  controlPort.send(dataPort.sendPort);
  controlPort.send((await dataPort.first) + 10);
  dataPort.close();
}

Future main() async {
  final controlPort = new ReceivePort();
  final bStream = controlPort.asBroadcastStream();
  await Isolate.spawn(isolateEntry, controlPort.sendPort);
  var dataPort = await bStream.first as SendPort;
  dataPort.send(100);
  print("Result: ${await bStream.first}");
  controlPort.close();
}
```


Isolates & ports

```
import "dart:isolate";

void cpuIntensiveWork(int delay) {
  final sw = Stopwatch()..start();
  while (sw.elapsedMilliseconds < delay);
}

void doWork(List args) {
  cpuIntensiveWork(5000);
  args[0].send(args[1] + 10);
}

Future main() async {
  final recPort = new ReceivePort();
  await Isolate.spawn(doWork, [recPort.sendPort, 100]);
  print('Result: ${await recPort.first}');
}
```

Isolates & ports

```
import "dart:isolate";

void cpuIntensiveWork(int delay) {
  final sw = Stopwatch()..start();
  while (sw.elapsedMilliseconds < delay);
}

class WorkDescr<TIn> {
  const WorkDescr(this.input, this.resultPort);
  final TIn input;
  final SendPort resultPort;
}

void isolateEntry(WorkDescr<int> input) {
  cpuIntensiveWork(5000);
  input.resultPort.send(input.input + 10);
}

Future main() async {
  final recPort = new ReceivePort();
  await Isolate.spawn(isolateEntry, WorkDescr<int>(100, recPort.sendPort));
  print("Result: ${await recPort.first}");
}
```

Isolates & ports

```
typedef WorkCallback<TIn, TOut> = TOut Function(TIn message);

class Payload<TIn, TOut> {
    const Payload(this.callback, this.input, this.resultPort);
    final WorkCallback<TIn, TOut> callback;
    final TIn input;
    final SendPort resultPort;
    TOut doWork() => callback(input);
}

void workEntrypoint<TIn, TOut>(Payload<TIn, TOut> work) {
    work.resultPort.send(work.doWork());
}

Future<TOut> doWork<TIn, TOut>(WorkCallback<TIn, TOut> callback, TIn input)
async {
    final resultPort = new ReceivePort();
    final descr = new Payload<TIn, TOut>(
        callback, input, resultPort.sendPort);
    await Isolate.spawn(workEntrypoint, descr);
    return await resultPort.first;
}
```

Isolates & ports

```
typedef ComputeCallback<Q, R> = R Function(Q message);

class Payload<Q, R> {
    const Payload(this.callback, this.input, this.resultPort);
    final ComputeCallback<Q, R> callback;
    final Q input;
    final SendPort resultPort;
    R apply() => callback(input);
}

void _spawn<Q, R>(Payload<Q, R> payload) =>
    payload.resultPort.send(payload.apply());

Future<R> compute<Q, R>(ComputeCallback<Q, R> callback, Q input) async {
    final resultPort = new ReceivePort();
    final payload = new Payload<Q, R>(callback, input, resultPort.sendPort);
    var isolate = await Isolate.spawn(_spawn, payload);
    var res = await resultPort.first;
    resultPort.close();
    isolate.kill();
    return res;
}
```

Isolates & ports

```
int bumpBy10(int v) => v + 10;

Future main() async {
  final result = await compute(bumpBy10, 100);

  print("Result: ${result}");
}
```



It's not new

Isolates in other languages

```
-module(tut15).  
-export([start/0, ping/2, pong/0]).  
ping(0, Pong_PID) ->  
    Pong_PID ! finished,  
    io:format("ping finished~n", []);  
ping(N, Pong_PID) ->  
    Pong_PID ! {ping, self()},  
    receive  
        pong ->  
            io:format("Received pong~n", [])  
    end,  
    ping(N - 1, Pong_PID).
```

```
pong() ->  
    receive  
        finished -> io:format("Pong finished~n", []);  
        {ping, Ping_PID} ->  
            io:format("Pong received ping~n", []),  
            Ping_PID ! pong,  
            pong()  
    end.  
start() ->  
    Pong_PID = spawn(tut15, pong, []),  
    spawn(tut15, ping, [3, Pong_PID]).
```

Takeaways

1. Isolates are a model of parallel execution,
2. Put severe constraints on what you can do,
3. You can think of them like separate processes (even separate VMs),
4. They communicate with messages only, so it's impossible to have data sharing.

Flutter

Flutter

1. It builds on top of DartVM, it **works exactly the same**,
2. Has a concept of ticks and frames (used to control rendering),
3. You can inject your logic in-between frames using `Ticker` and `SchedulerBinding` mixin,
4. Other than that there are no differences in `Futures`, `Streams`, `Zones` and `Isolates` mechanisms,
5. `compute` (shown earlier) is available in Flutter's foundation package.

Summary

1. Dart uses a single-threaded model of execution,
2. But remember that it does not mean it uses single native thread, you just don't see them,
3. To model asynchrony you can use Futures and Streams,
4. Parallel execution is done using Isolates that can only communicate with messages through ports,
5. It's really fun to dig into DartVM. ;)



Thanks!