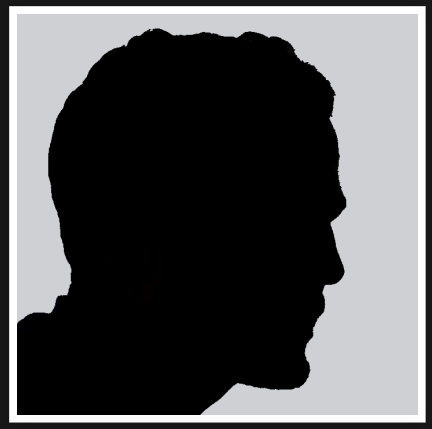
FROM HTTP TO KAFKA-BASED MICROSERVICES

Wojciech Rząsa, FLYR Poland

@wrzasa



FLYR

RZESZÓW RUBY USER GROUP

ABOUT ME

- Informatics specialist by passion and by profession
- 15 years of academic work
- PhD but primarily an engineer
- FLYR Inc. http://flyrlabs.com
- Distributed systems
- Rzeszow Ruby User Group http://rrug.pl

FLYR

- Revenue management system for airlines
- Offices in
  - San Francisco, USA (PST)
  - Kraków, Poland (CEST)
- Machine Learning
- Microservices
- Python
- GCloud
- Kubernetes

FLYR DEVS ON PYCON CZ

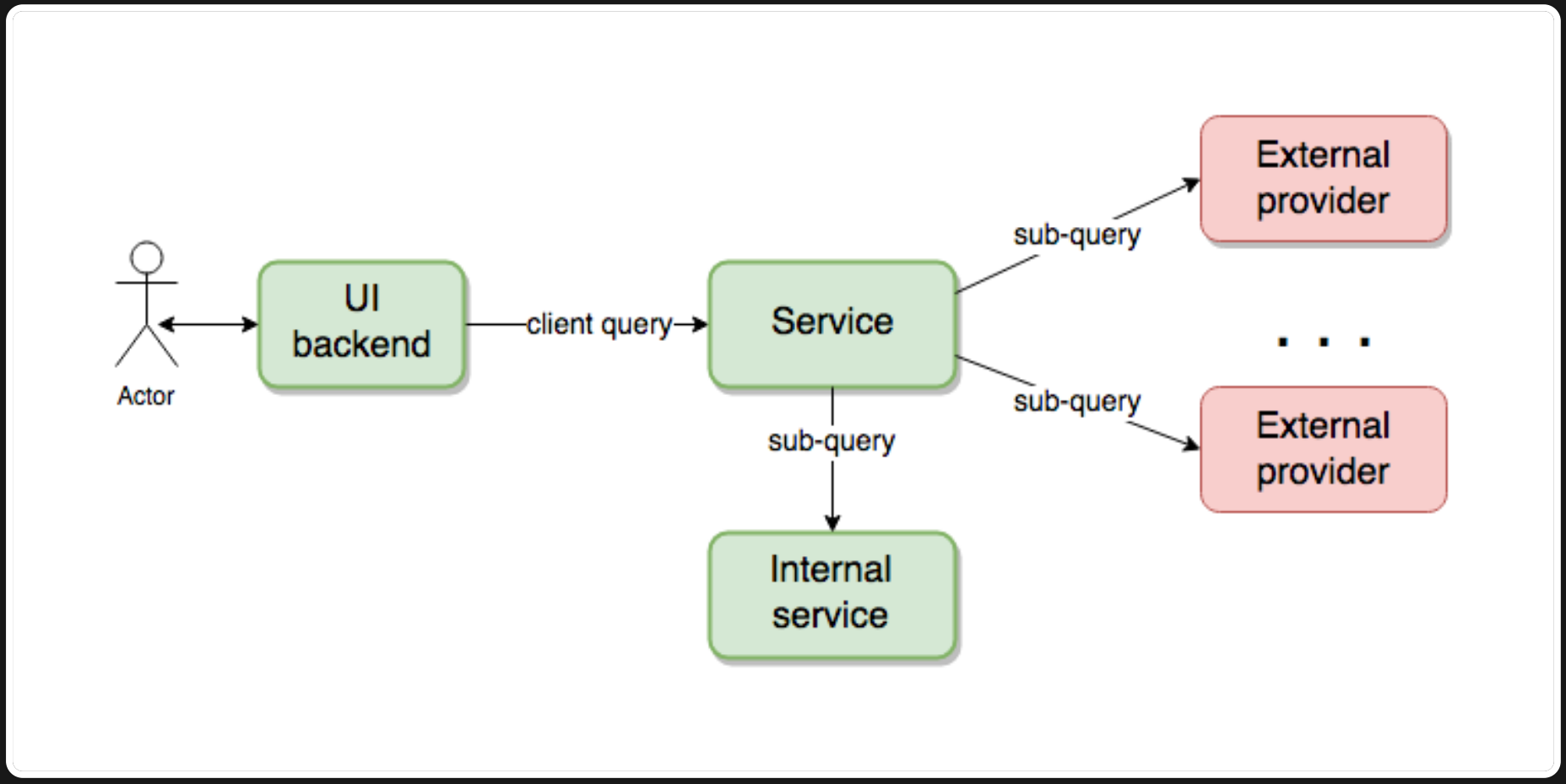
IN FLYR MICROSERVICES

- IPC based on HTTP

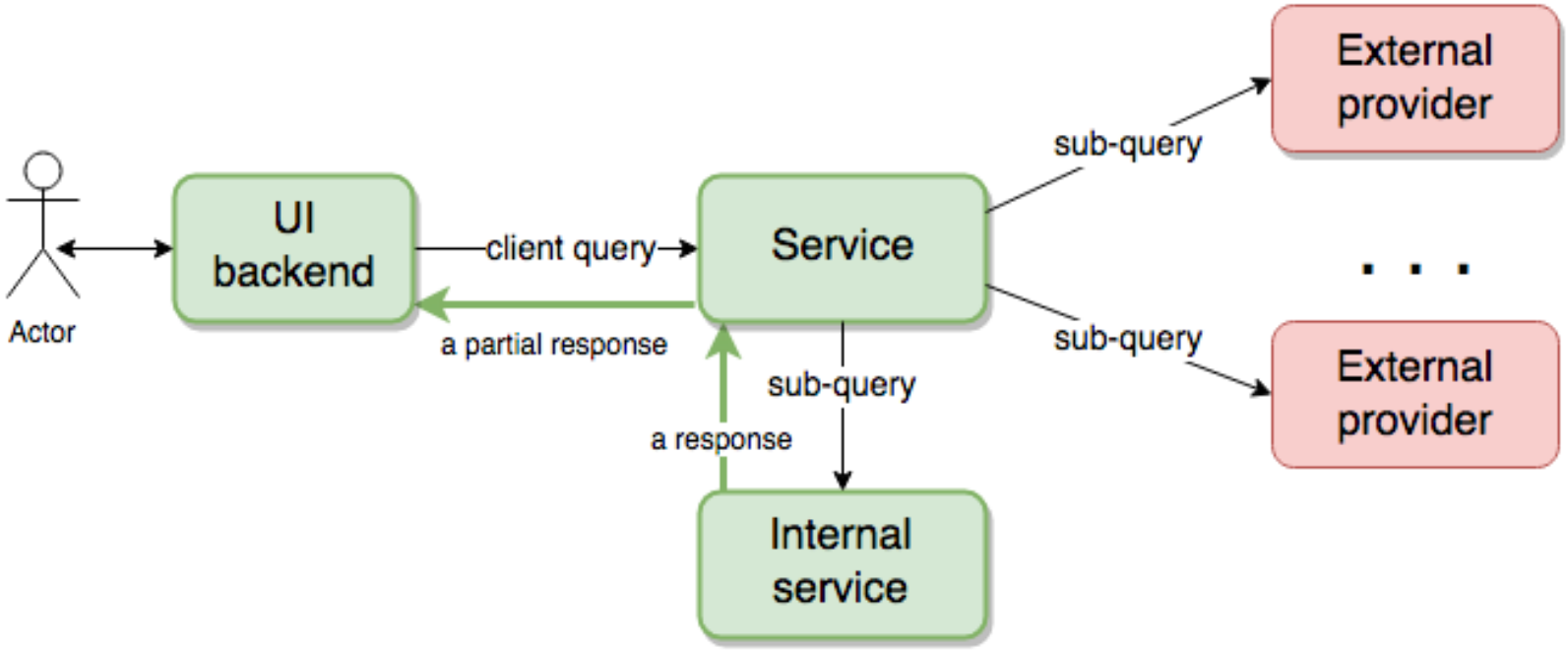
IN FLYR MICROSERVICES

- IPC based on HTTP
- New requirements for eCommerce use case

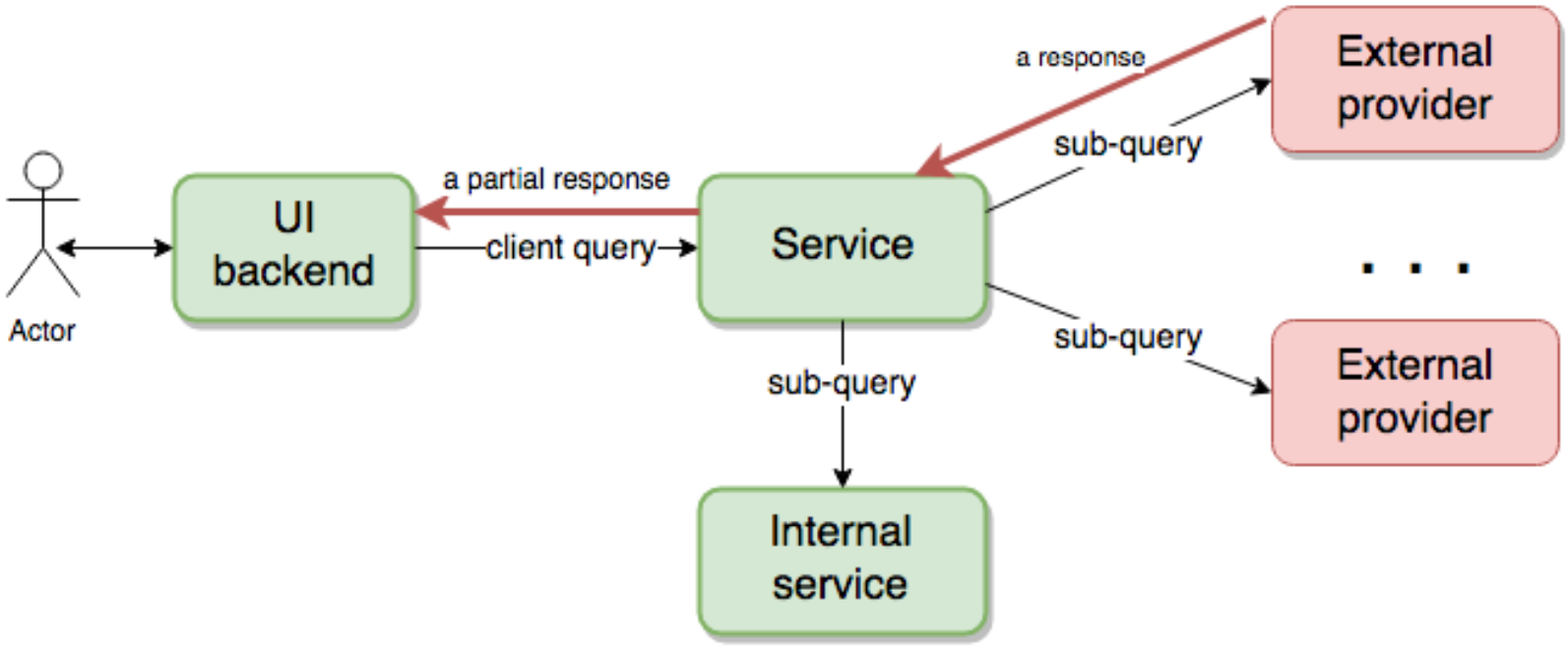
FAN-OUT REQUESTS



PARTIAL RESPONSES



PARTIAL RESPONSES



IN FLYR MICROSERVICES

- IPC based on HTTP
- New requirements for eCommerce use case
  - partial responses

IN FLYR MICROSERVICES

- IPC based on HTTP
- New requirements for eCommerce use case
  - partial responses
  - performance

OK, LET'S SWITCH FROM HTTP TO... A... MQ?

BUT...

BUT...

- We have HTTP-based infrastructure

BUT...

- We have HTTP-based infrastructure
- We have HTTP developers experience and habits

BUT...

- We have HTTP-based infrastructure
- We have HTTP developers experience and habits
- We lack experience with MQ-based IPC

BUT...

- We have HTTP-based infrastructure
- We have HTTP developers experience and habits
- We lack experience with MQ-based IPC
- We need to do it well ;-)

WITH MQ WE GET

- Flexibility
- Reliability
- Scalability
- Robustness

NEW OPPORTUNITIES...

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions
- Incorrect broker choice

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions
- Incorrect broker choice
- Incorrect driver for the correct broker

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions
- Incorrect broker choice
- Incorrect driver for the correct broker
- Incorrect usage patterns for the correct driver

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions
- Incorrect broker choice
- Incorrect driver for the correct broker
- Incorrect usage patterns for the correct driver
- Incorrect usage patterns for the correct broker

NEW OPPORTUNITIES... TO MAKE NEW MISTAKES

- Concurrency issues
- Race conditions
- Incorrect broker choice
- Incorrect driver for the correct broker
- Incorrect usage patterns for the correct driver
- Incorrect usage patterns for the correct broker
- . . .

LET'S CONTAIN THE RISKS IN ONE PLACE

LET'S CONTAIN THE RISKS IN ONE PLACE

(A LIBRARY)

AND CALL THIS PLACE

`async_calls`

AND CALL THIS PLACE

`async_calls`

(FOR THE LACK OF BETTER CONCEPT)

A LIBRARY THAT

- meets functional requirements
- for developers, resembles HTTP where possible
- uses a MQ broker for communication

FOR MAINTAINERS

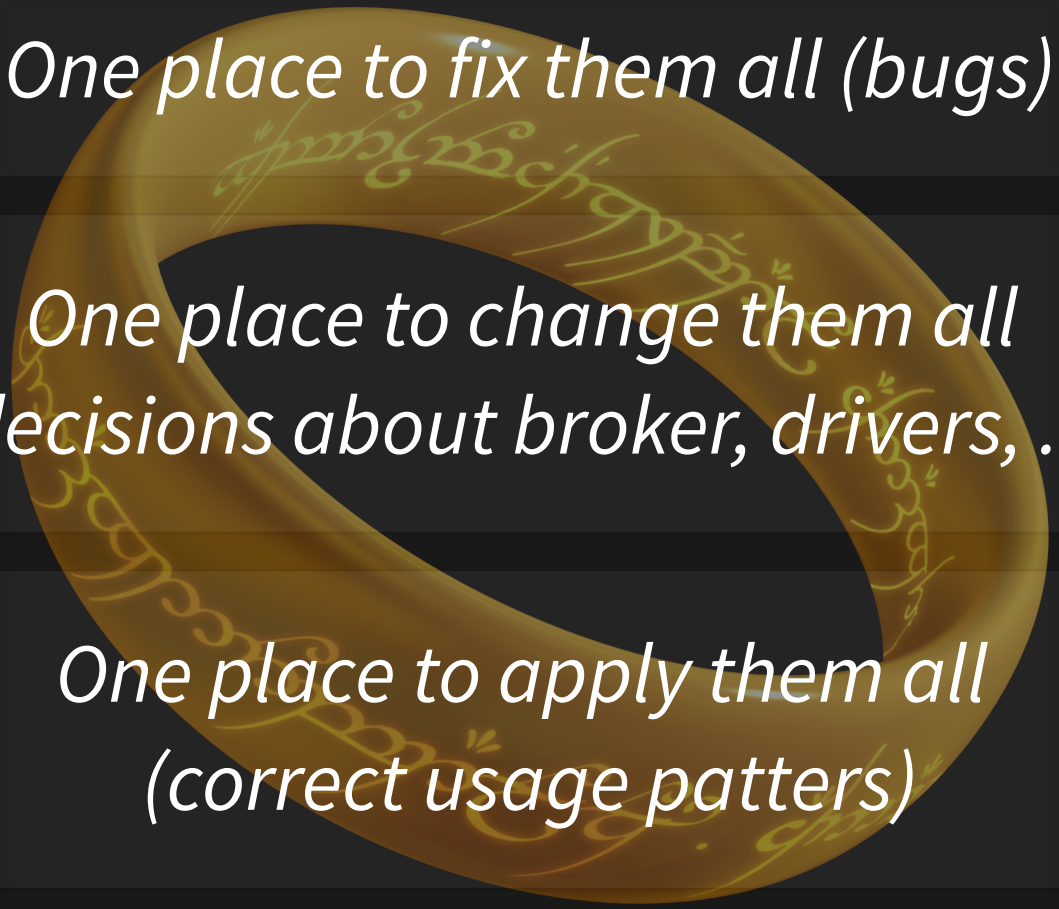
THE SAURON ADVANTAGE :)

FOR MAINTAINERS

THE SAURON ADVANTAGE :)

FOR MAINTAINERS

THE SAURON ADVANTAGE :)



One place to fix them all (bugs)

One place to change them all
(decisions about broker, drivers, ...)

One place to apply them all
(correct usage patters)

FOR DEVELOPERS

- New complexity is hidden
- Lower entry barrier

DECISIONS

DECISIONS

- Message Broker – Kafka
  - performance
  - persistence

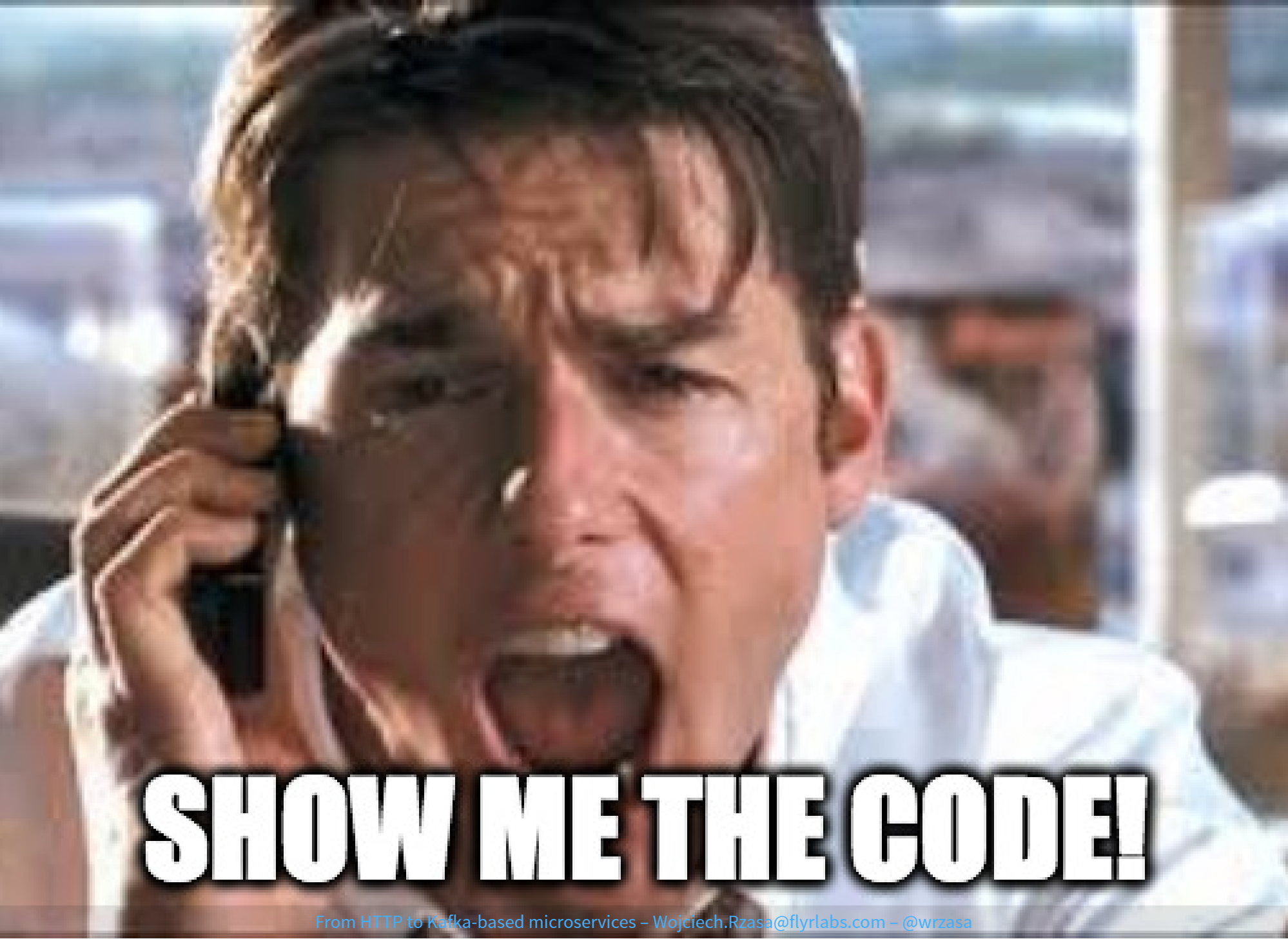
DECISIONS

- Message Broker – Kafka
  - performance
  - persistence
- Kafka driver – `confluent-kafka`
  - performance
  - supported by Confluent

ASSUMPTIONS

- Make it simple – provide just IPC
- Library, not framework approach
- Make it testable
  - manually (`curl`-like tool)
  - automatically (reasonable mocks)
- Make it resemble Flask?

TALK IS CHEAP!


SHOW ME THE CODE!

CREATE AN APPLICATION OBJECT

```
from async_calls import AsyncCalls

async_calls = AsyncCalls('a-money-broker')
```

CREATE AN APPLICATION OBJECT

```
from async_calls import AsyncCalls

async_calls = AsyncCalls('a-money-broker')
```

CREATE A BASIC ENDPOINT

```
@async_calls.server.callback_for('/show-me-the-money')
def show_me_the_money(request):
    for i in range(1,5):
        payload = f"Response {i} for call: {request.id}"
        response = request.create_response(payload)
        async_calls.server.send(response)
        time.sleep(1)
```

CREATE AN APPLICATION OBJECT

```
from async_calls import AsyncCalls

async_calls = AsyncCalls('a-money-broker')  # a service ID
```

CREATE A BASIC ENDPOINT

```
@async_calls.server.callback_for('/show-me-the-money')
def show_me_the_money(request):
    for i in range(1,5):
        payload = f"Response {i} for call: {request.id}"
        response = request.create_response(payload)
        async_calls.server.send(response)
        time.sleep(1)
```

CREATE AN APPLICATION OBJECT

```
from async_calls import AsyncCalls

async_calls = AsyncCalls('a-money-broker')  # a service ID
```

CREATE A BASIC ENDPOINT

```
@async_calls.server.callback_for('/show-me-the-money')
def show_me_the_money(request): # ^^^^^^^^ an endpoint name
    for i in range(1,5):
        payload = f"Response {i} for call: {request.id}"
        response = request.create_response(payload)
        async_calls.server.send(response)
        time.sleep(1)
```

CREATE A BASIC CLIENT

```
request = async_calls.client.new_message(
    destination_service_id: 'a-money-broker',
    target_endpoint: '/show-me-the-money',
    request_payload
)
async_calls.client.send(request)
```

CREATE A BASIC CLIENT

```
@async_calls.client.callback_for(
                    'a-money-broker', '/show-me-the-money')
def the_money_handler(response):
    logger.info(
        f"Got: {response.id} for: {response.referenced_id}"
    )

request = async_calls.client.new_message(
    destination_service_id: 'a-money-broker',
    target_endpoint: '/show-me-the-money',
    request_payload
)
async_calls.client.send(request)
```

TO START LISTENING

(CLIENT AND SERVER)

```
async_calls.listen()
```

WHAT WE HAVE THEN?

WHAT WE HAVE THEN?

- Server — event-driven (like HTTP)

WHAT WE HAVE THEN?

- Server — event-driven (like HTTP)
- Client — non-blocking, event-driven (unlike HTTP)

WHAT WE HAVE THEN?

- Server — event-driven (like HTTP)
- Client — non-blocking, event-driven (unlike HTTP)
- One request — any number of responses

WHAT WE HAVE THEN?

- Server — event-driven (like HTTP)
- Client — non-blocking, event-driven (unlike HTTP)
- One request — any number of responses
- A single process can be a server and a client

HOW DO WE TEST THIS!?

async_calls

HAS A TESTING MODE

```
# setup testing mode
from async_calls import AsyncCalls
AsyncCalls.testing = True
```

async_calls

HAS A TESTING MODE

```
# setup testing mode
from async_calls import AsyncCalls
AsyncCalls.testing = True

# import your app
from ..async_endpoint import async_calls
```

async_calls

HAS A TESTING MODE

```
# setup testing mode
from async_calls import AsyncCalls
AsyncCalls.testing = True

# import your app
from ..async_endpoint import async_calls

# ensure you reset state between tests
@pytest.fixture(autouse=True)
def reset_async_calls_test_mode():
    async_calls.test_mode_reset()
```

TESTING A SERVER

DOES IT RESPOND CORRECTLY?

```
# Send a test request (using test_client)
request = async_calls.test_client.new_message(
    'a-money-broker', '/show-me-the-money', 'A money request'
)
async_calls.test_client.send(request)
```

TESTING A SERVER

DOES IT RESPOND CORRECTLY?

```
# Send a test request (using test_client)
request = async_calls.test_client.new_message(
    'a-money-broker', '/show-me-the-money', 'A money request'
)
async_calls.test_client.send(request)

# Check what response were received (by test_client)
responses = async_calls.test_client.received_responses()
```

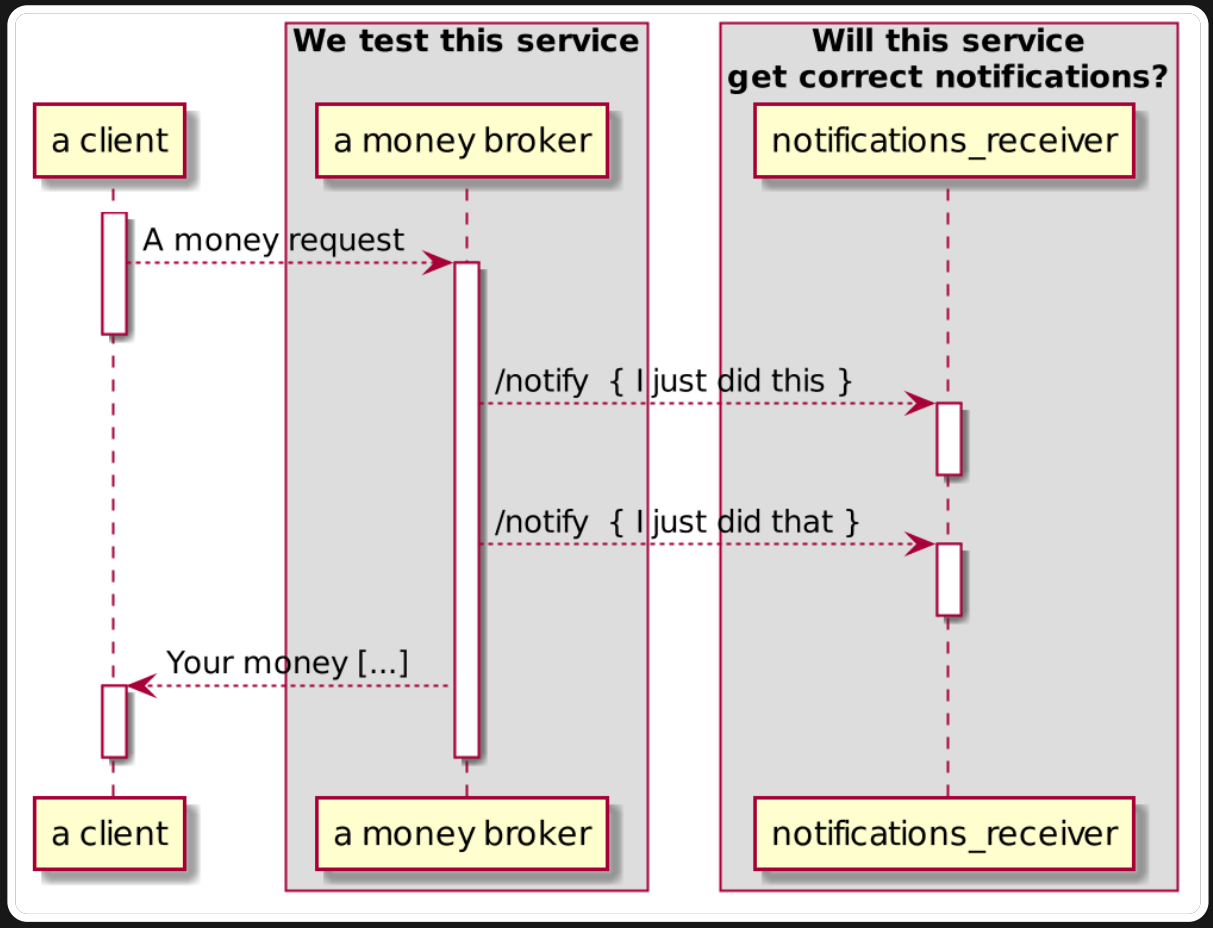
TESTING A SERVER

DOES IT RESPOND CORRECTLY?

```
# Send a test request (using test_client)
request = async_calls.test_client.new_message(
    'a-money-broker', '/show-me-the-money', 'A money request'
)
async_calls.test_client.send(request)

# Check what response were received (by test_client)
responses = async_calls.test_client.received_responses()

# Set your expectations
expected_payloads = ["the money", "you expect"]

received_payloads = [m.payload for m in responses]
assert sorted(expected_payloads) == sorted(received_payloads)
```

TESTING A CLIENT

DOES IT SEND EXPECTED REQUESTS?



TESTING A CLIENT

DOES IT SEND EXPECTED REQUESTS?

```
# Unit tests for 'a money broker' service
# register endpoint in test_server
async_calls.test_server.register_endpoint(
    'notifications_receiver', '/notifiy'
)
```

TESTING A CLIENT

DOES IT SEND EXPECTED REQUESTS?

```
# Unit tests for 'a money broker' service
# register endpoint in test_server
async_calls.test_server.register_endpoint(
    'notifications_receiver', '/notifiy'
)

# some testing actions that should trigger messages to the
# '/notify' endpoint of 'notifications_receiver' service
```

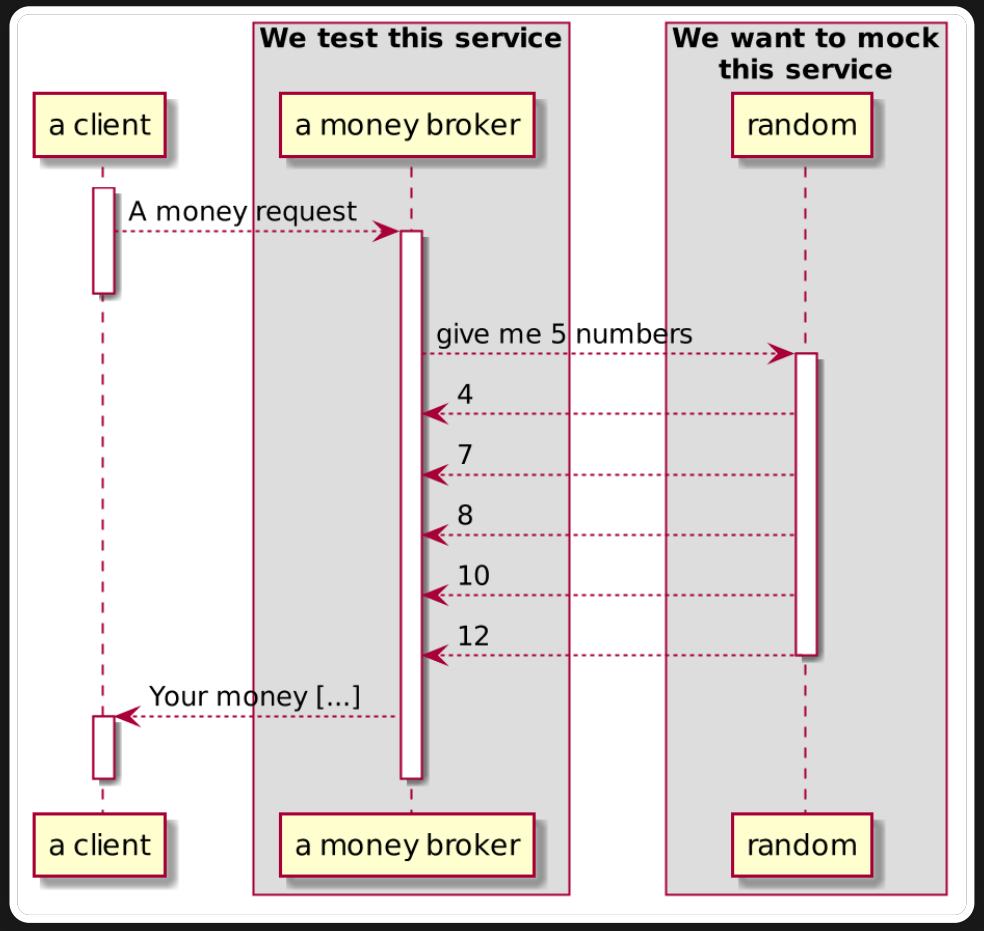
TESTING A CLIENT

DOES IT SEND EXPECTED REQUESTS?

```
# Unit tests for 'a money broker' service
# register endpoint in test_server
async_calls.test_server.register_endpoint(
    'notifications_receiver', '/notifiy'
)

# some testing actions that should trigger messages to the
# '/notify' endpoint of 'notifications_receiver' service

rcved = async_calls.test_server.received_requests('/notify')
# here you can assert that all required messages were
# received by the '/notify' endpoint
```

TESTING CLIENT

FAKING SERVER'S RESPONSE



TESTING CLIENT

FAKING SERVER'S RESPONSE

```
# Unit tests for 'a money broker' service
# Create a fake random number service
def fake_responses(request):
    for i in [4, 7, 8, 10, 12]:
        yield request.create_response(i)
```

TESTING CLIENT

FAKING SERVER'S RESPONSE

```
# Unit tests for 'a money broker' service
# Create a fake random number service
def fake_responses(request):
    for i in [4, 7, 8, 10, 12]:
        yield request.create_response(i)

# register endpoint with generator in test_server
async_calls.test_server.register_endpoint(
    'random', '/get_values', fake_responses
)
```

TESTING SUMMARY

- Tools out-of-the-box
- Calls made on stack, deterministic tests
- No MQ broker required for unittests
- No need to think about IPC details when implementing tests

MANY MORE FEATURES

MANY MORE FEATURES

- before send and before receive hooks (e.g. for validations)

MANY MORE FEATURES

- before send and before receive hooks (e.g. for validations)
- endpoint context managers (e.g. for performance measurements)

MANY MORE FEATURES

- before send and before receive hooks (e.g. for validations)
- endpoint context managers (e.g. for performance measurements)
- endpoint error handlers

MANY MORE FEATURES

- before send and before receive hooks (e.g. for validations)
- endpoint context managers (e.g. for performance measurements)
- endpoint error handlers
- Kubernetes healthcheck

MANY MORE FEATURES

- before send and before receive hooks (e.g. for validations)
- endpoint context managers (e.g. for performance measurements)
- endpoint error handlers
- Kubernetes healthcheck
- CLI curl-like client

ANY DRAWBACKS?

- Hiding complexity we hide opportunities...
- ...not only to make new errors
- e.g. no Kafka Streams via `async_calls`

HOW DID IT SAVE US?

- Concurrency issues
- Race conditions
- Incorrect broker choice
- Incorrect driver for the correct broker
- Incorrect usage patterns for the correct driver
- Incorrect usage patterns for the correct broker
- . . .

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated
- Support for one-way communication

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated
- Support for one-way communication
- More complex use cases require more attention

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated
- Support for one-way communication
- More complex use cases require more attention
- Services are easily testable

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated
- Support for one-way communication
- More complex use cases require more attention
- Services are easily testable
- Standard project-wide layer for asynchronous IPC

SUMMARY

- Switching from HTTP to `async_calls`
  - Server is straightforward
  - Client is not complicated
- Support for one-way communication
- More complex use cases require more attention
- Services are easily testable
- Standard project-wide layer for asynchronous IPC
- A number of small but useful bonuses