brigad

Let's work

Resilience: why, how, what?

Aymeric Beaumet, Software Architect at Brigad

resilience *(noun)*: the ability of a substance to return to its usual shape after being bent, stretched, or pressed

Cambridge Dictionary

A home in Gilchrist, Texas, *designed* to resist flood waters *survived* Hurricane Ike in 2008. Credits: FEMA/Joselyne Augustino

Why? I want my construction to resist in extreme conditions for a reasonable amount of time.

How? Identify the risks. Adapt in consequence.

What? Robust foundation. Elastic structures. Strongly fixed roof. etc.

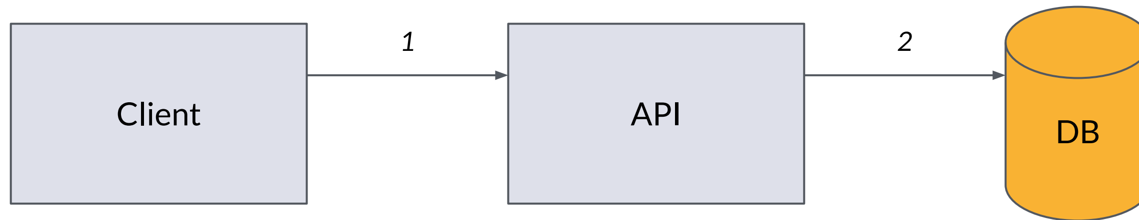
Similar why.

Similar how.

Let’s focus on the what.

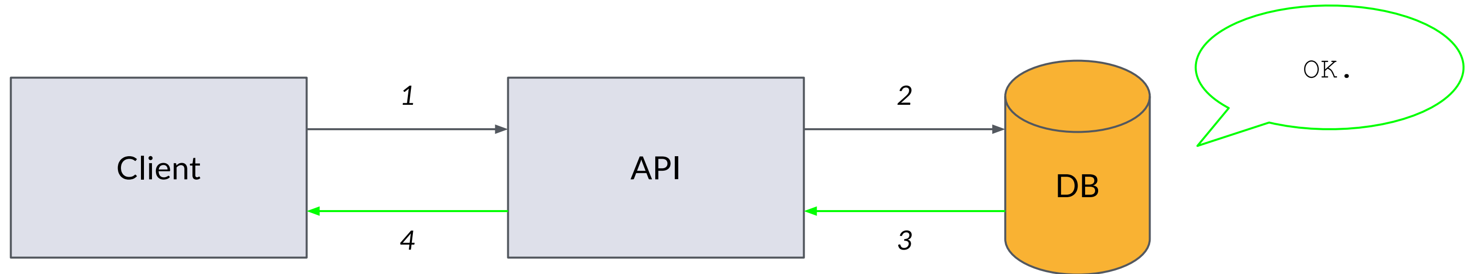
Scenario 1

" Update email address "



Scenario 1

“ Update email address ”

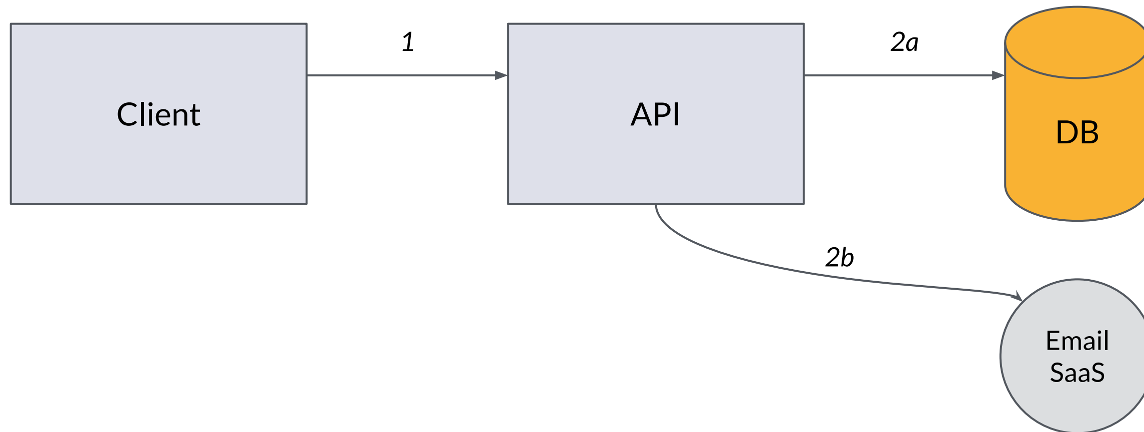


```
const app = require('./express')
const { User } = require('./sequelize')

app.patch('/users/:id', async (req, res) => {
  try {
    // Update user in DB
    await User.update(req.body, {
      where: { id: req.params.id }
    })
    return res.status(200).send()
  } catch (error) {
    return res.status(500).send()
  }
})
```

The client is responsible for the completion of the action it has initiated.

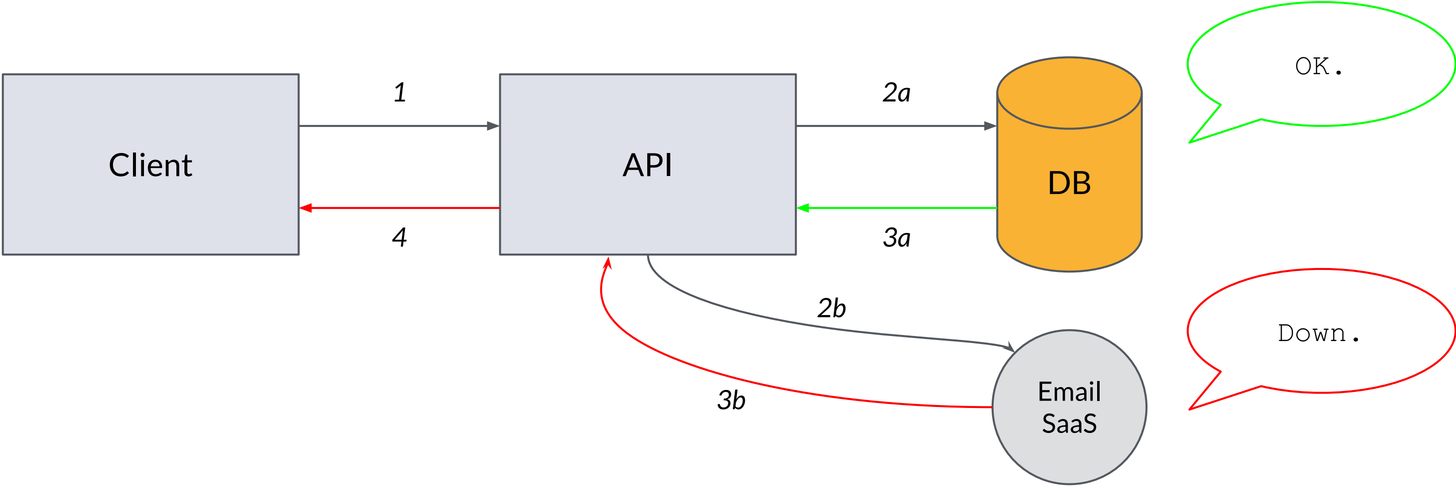
Scenario 2

“ I want to send a confirmation email ”

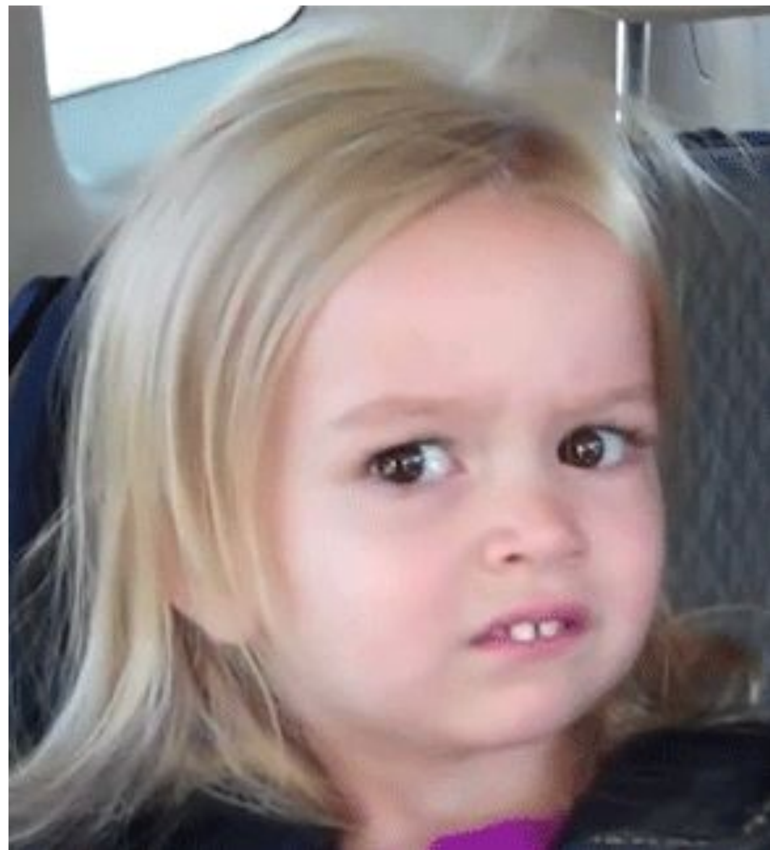


Scenario 2

“ I want to send a confirmation email ”



```javascript
const app = require('./express')
const { sequelize, User } = require('./sequelize')
const email = require('./email')

app.patch('/users/:id', async (req, res) => {
  try {
    // Update user in DB
    const user = await User.update(req.body, {
      where: { id: req.params.id },
      returning: true,
      plain: true,
    })
    // Send email
    if (user.email !== req.params.email) {
      await email.confirm(user.email) // UNSAFE
    }
    return res.status(200).send()
  } catch (error) {
    return res.status(500).send()
  }
})
```



```
const app = require('./express')
const { sequelize, User } = require('./sequelize')
const email = require('./email')

app.patch('/users/:id', async (req, res) => {
  try {
    // Wrap in a transaction
    await sequelize.transaction(async (t) => {
      // Update user in DB
      const user = await User.update(req.body, {
        where: { id: req.params.id },
        returning: true,
        plain: true,
        transaction: t,
      })
      // Send email
      if (user.email !== req.params.email) {
        await email.confirm(user.email) // SAFE
      }
      return res.status(200).send()
    })
  } catch (error) {
    return res.status(500).send()
  }
})
```

Great!

But should it really fail?

Ownership delegation

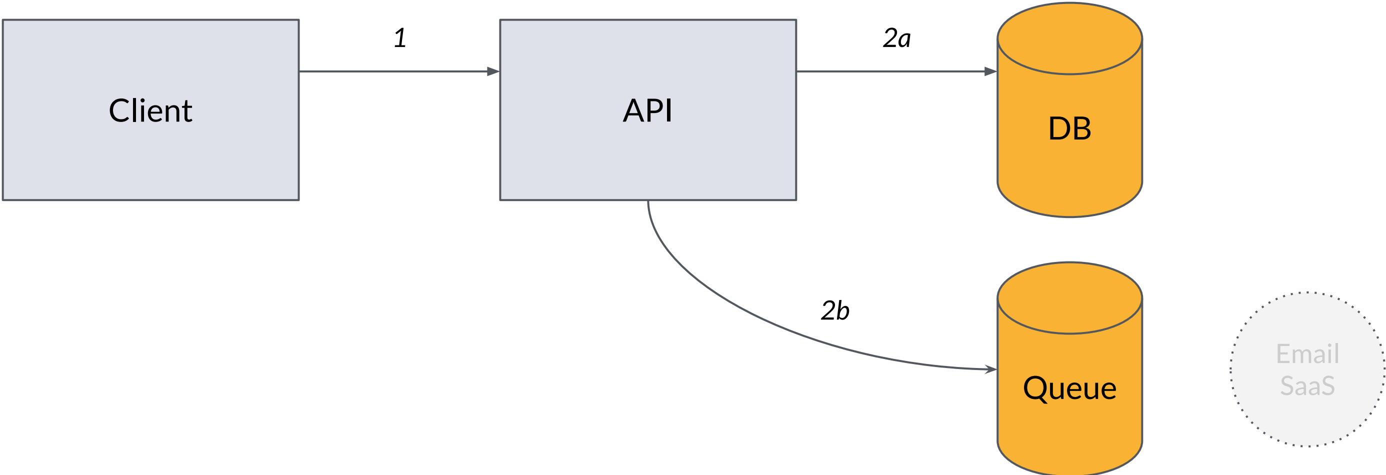
1/ It's the initiator responsibility to wait and retry until the ownership is delegated.

2/ The ownership is delegated once the initiator's requests succeeds.

3/ Leverage stateful services (Database, Queue, etc)

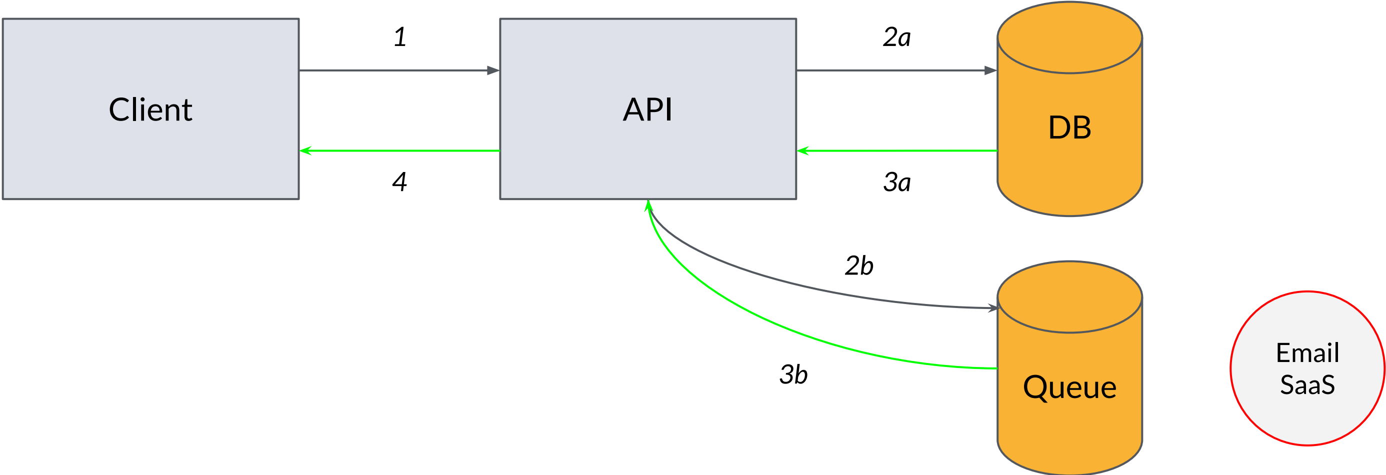
Scenario 3

“ I want my clients to only fail when they’re responsible ”



Scenario 3

“ I want my clients to only fail when they’re responsible ”



```
const app = require('./express')
const { sequelize, User } = require('./sequelize')
const queue = require('./queue')

app.patch('/users/:id', async (req, res) => {
  try {
    // Wrap in a transaction
    await sequelize.transaction(async (t) => {
      // Update user in DB
      const user = await User.update(req.body, {
        where: { id: req.params.id },
        returning: true,
        plain: true,
        transaction: t,
      })
      // Push for later
      if (user.email !== req.params.email) {
        await queue.push(user.email)
      }
      return res.status(200).send()
    })
  } catch (error) {
    return res.status(500).send()
  }
})
```

```
const email = require('./email')
const queue = require('./queue')

async function work() {
  while (true) {
    try {
      const message = await queue.receiveMessage().promise() // blocking
      await email.send(message)
      await queue.deleteMessage(message).promise() // UNSAFE
    } catch (error) {
      console.error(error)
      // let the message become visible again in the queue
    }
  }
}

work()
```

Idempotence

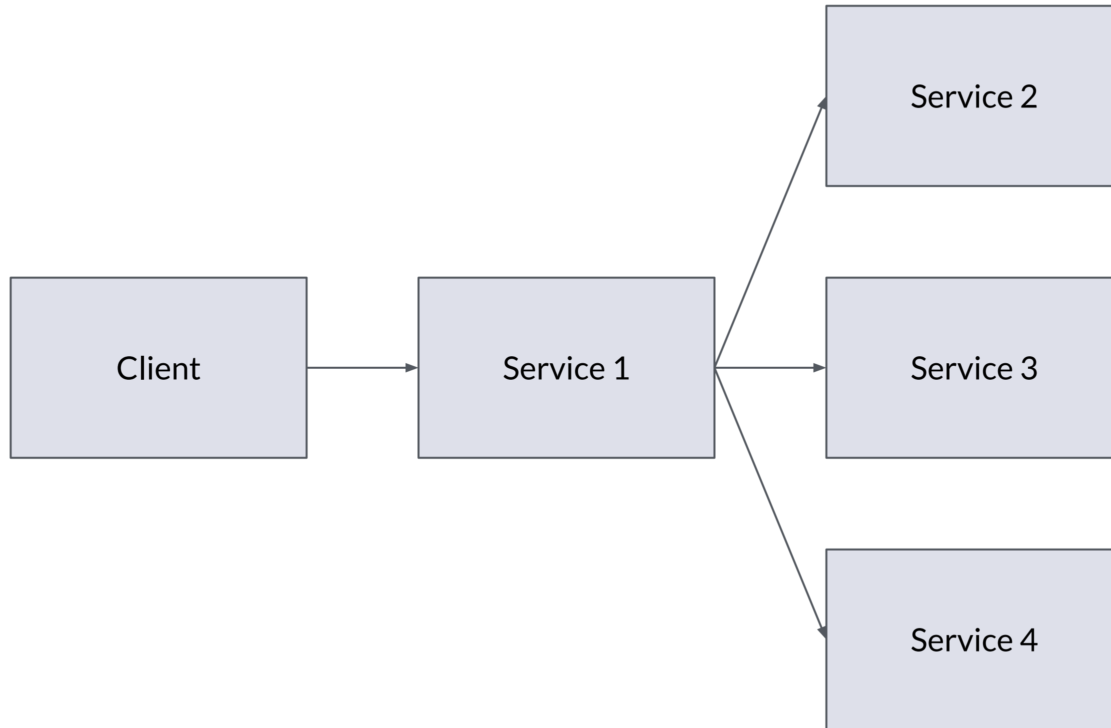
An action which can be run several times without changing the result.

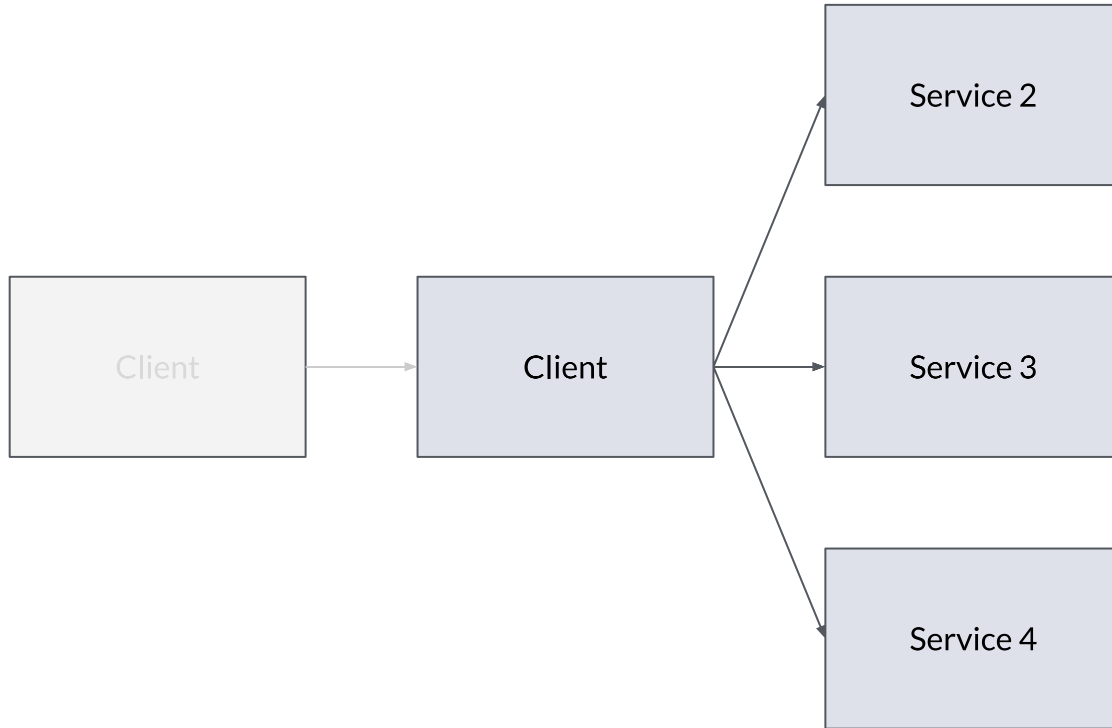
e.g.: considering HTTP REST, the *HEAD*, *GET*, *PUT*, *DELETE*, *OPTIONS* methods are idempotent by design.

```
const app = require('./express')
const { sequelize, User } = require('./sequelize')
const queue = require('./queue')

app.patch('/users/:id', async (req, res) => {
  try {
    // Wrap in a transaction
    await sequelize.transaction(async (t) => {
      // Update user in DB
      const user = await User.update(req.body, {
        where: { id: req.params.id },
        returning: true,
        plain: true,
        transaction: t,
      })
      // Push for later
      if (user.email !== req.params.email) {
        await queue.push(user.email)
      }
      return res.status(200).send()
    })
  } catch (error) {
    return res.status(500).send()
  }
})
```

```
const email = require('./email')
const queue = require('./queue')
const { sequelize, EmailStatus } = require('./sequelize')

async function work() {
  while (true) {
    try {
      await sequelize.transaction(async function(t) {
        const message = await queue.receiveMessage().promise() // blocking
        const hash = hashMessage(message)
        await EmailStatus.insert({ id: hash }, { transaction: t}) // guarantee unicity
          .then(() => email.send(message))
        await queue.deleteMessage(message).promise().catch(error => console.warn(error))
      })
    } catch (error) {
      console.error(error)
      // let the message become visible again in the queue
    }
  }
}

work()
```

Does it work in a distributed system?


Client
Service 1
Service 2
Service 3
Service 4


Service 2
Client
Client
Service 3
Service 4

Conclusion

- Ownership delegation is a mindset
- Leverage persistent services (DB, Queue, ...)
- Always wonder what happens if *this* fails
- Be idempotent when you can
- A resilient architecture is a collaboration of atomic systems
- Can be applied at every level of the architecture

Questions?

- Concurrency in a distributed system
- Circuit Breaker (https://martinfowler.com)
- AZ redundancy (https://docs.aws.amazon.com)