

API evolution without versioning

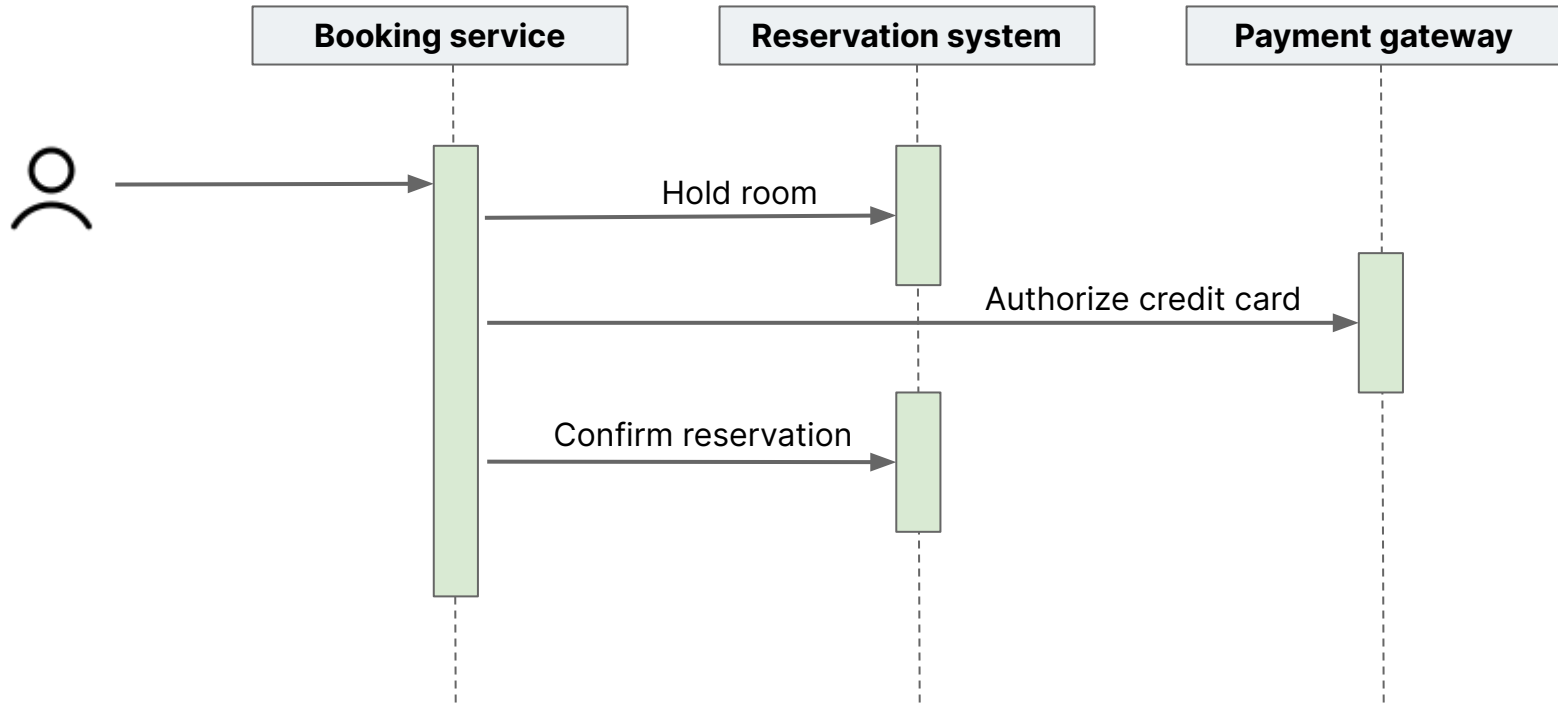
Brandon Byars

Head of Technology

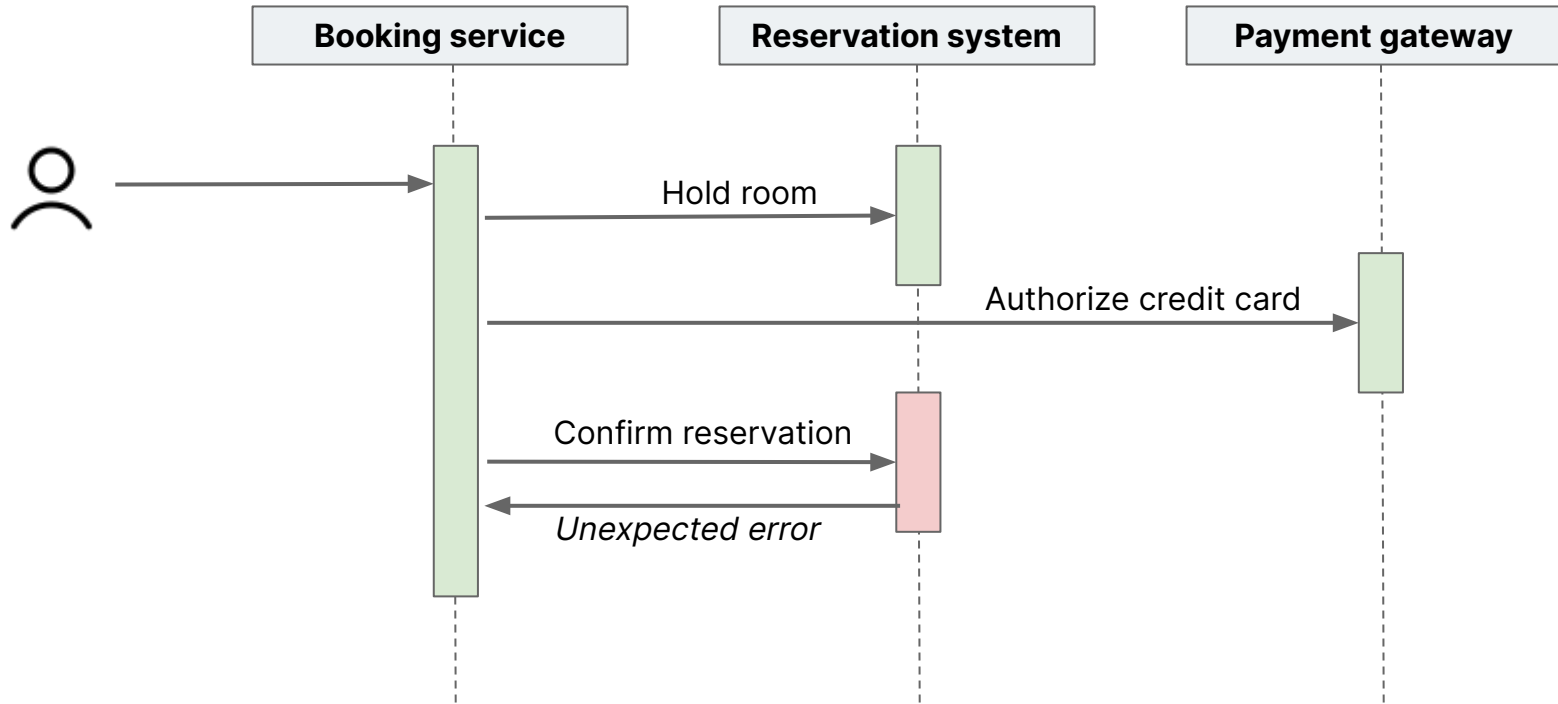
bbyars@thoughtworks.com

@BrandonByars

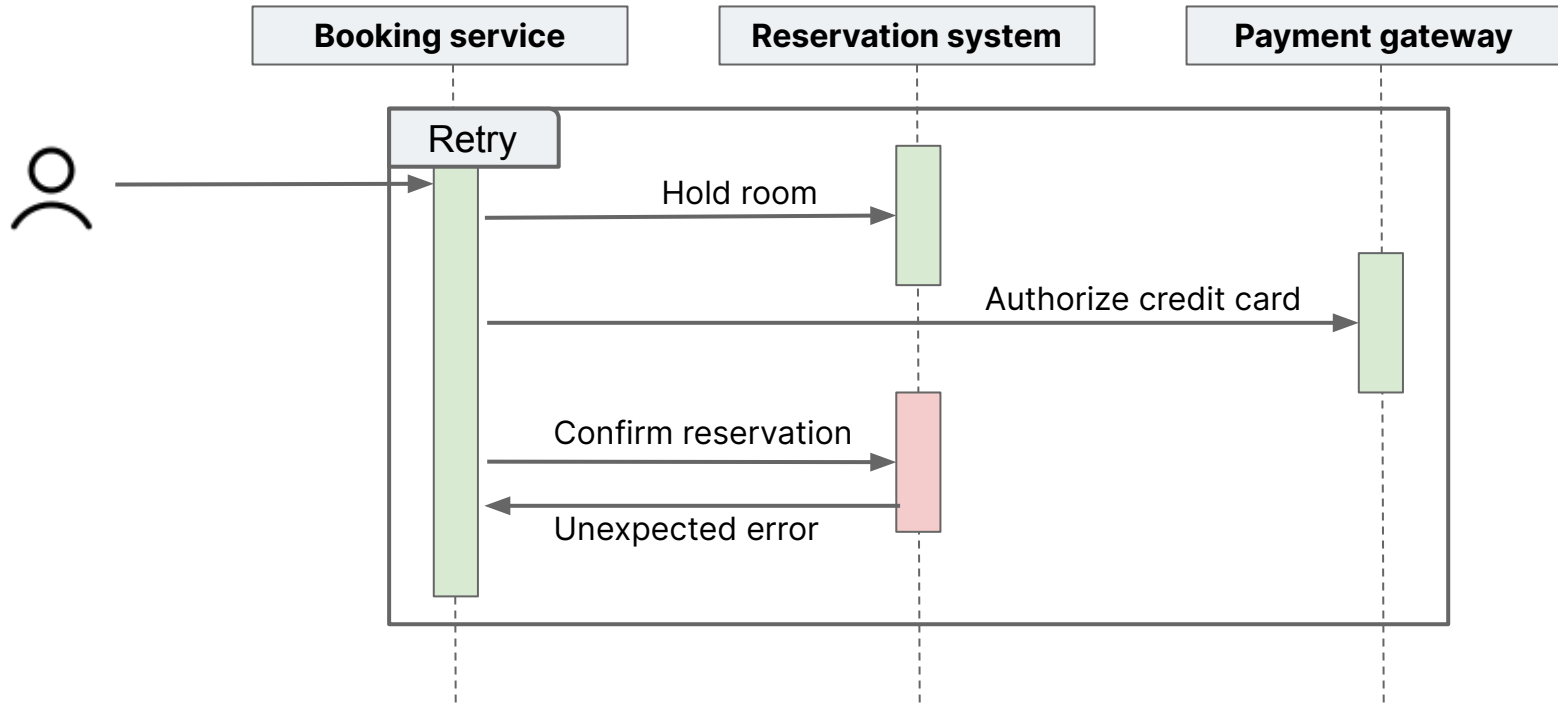
Booking a cruise



Booking a cruise



Booking a cruise



About me

Experiences

- Led multiple digital platform engagements with a focus on API strategy

Books

- Testing microservices with mountebank

Articles on martinfowler.com

- Enterprise integration using REST
- You can't buy integration
- *API evolution without versioning*



MOVE FAST AND BREAK THINGS*

* (THAT OTHER PEOPLE
HAVE TO FIX)

Some people, when confronted with a problem, think “I know, I’ll use versioning.” Now they have 2.1.0 problems.

An adaptation of Jamie Zawinski’s famous dig on regular expressions

Contract specifications as promises

```
paths:
  /audio/volume:
    put:
      requestBody:
        required: true
      content:
        application/json:
          schema:
            type: integer
            minimum: 0
            maximum: 11
            description:
              Current volume for all audio output.
              0 means no audio output (mute). 10 is the maximum value. 11 enables
              the overdrive system (danger!).
              When set to 0 all other audio settings have no effect.
              ...
```

So why the word
promise?

...

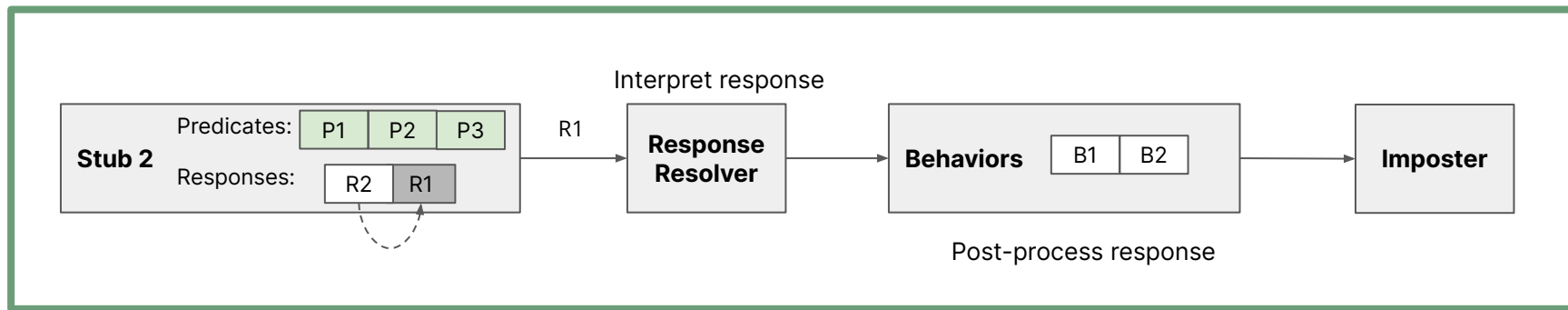
The term does not have
the arrogance or hubris of
the more frequently used
guarantee, and that is
good.

- Mark Burgess

Following the evolution of a complex API



mountebank (<https://www.mbtest.org>) is a multi-protocol service virtualization tool controllable through a REST API



Evaluating options from a consumer's perspective

All strategies will be evaluating against the following criteria. There is one criteria missing, which is often the primary one used for decision making: *implementation complexity*.

1.

Obviousness

2.

Elegance

3.

Stability

Evolution patterns

Change by addition

Add latency to the response

```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  }
}
```



```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  },
  "_behaviors": {
    "wait": 500
  }
}
```

Obviousness



Elegance



Stability



Multi-typing

Allow the latency to be determined dynamically

```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  },
  "_behaviors": {
    "wait": 500
  }
}
```



```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  },
  "_behaviors": {
    "wait": "() => Math.random() * 1000"
  }
}
```

Obviousness



Elegance



Stability



Upcasting

Allow a pipeline of shell transformations instead of just one

```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  },
  "_behaviors": {
    "wait": 500,
    "shellTransform": "ruby ./time.rb"
  }
}
```



```
{
  "is": {
    "statusCode": 500,
    "body": "Error!!!"
  },
  "_behaviors": {
    "wait": 500,
    "shellTransform": ["ruby ./time.rb",
"python lookup.py"]
  }
}
```

Obviousness



Elegance



Stability



Isn't that a breaking change?

Of course it is!

Versioning pushes the cost of the breaking change to the consumers.

Upcasting absorbs the cost of the breaking change in the producer.

```
compatibility.upcast(request);
```

```
function upcast (request) {
  upcastShellTransformToArray(request);
}

function upcastShellTransformToArray (request) {
  (request.stubs || []).forEach(stub => {
    (stub.responses || []).forEach(response => {
      if (response._behaviors &&
        response._behaviors.shellTransform &&
        typeof
        response._behaviors.shellTransform === 'string') {
        response._behaviors.shellTransform =
          [response._behaviors.shellTransform];
      }
    });
  });
}
```

Nested upcasting

Allow a pipeline of shell transformations instead of just one

```
{
  "is": { ... },
  "_behaviors": {
    "wait": 500,
    "shellTransform": ["...", "..."],
    "lookup": [{ ... }, { ... }],
    "copy": [{ ... }, { ... }],
    "decorate": "...",
    "repeat": 3
  }
}
```



```
{
  "is": " { ... }",
  "repeat": 3,
  "behaviors": [
    { "copy": { ... } },
    { "decorate": "...",
      "lookup": { ... },
      "shellTransform": "...",
      "decorate": "...",
      "wait": 500,
      "shellTransform": "..." }
  ]
}
```

Obviousness



Elegance



Stability



Implementing nested upcasts

The trick is to simply execute the upcasts in chronological order of change

```
compatibility.upcast(request);
```

```
function upcast (request) {  
  upcastShellTransformToArray(request);  
  upcastBehaviorsToArray(request);  
}
```

Downcasting

Replace complicated function signature with a Parameter Object

```
{  
  "inject": "function (request,  
    deprecatedLocalState, logger, callback,  
    imposterState) { ... }"  
}
```



```
{  
  "inject": "function (config) { ... }"  
}
```

Obviousness



Elegance



Stability



Implementing downcasting

Downcasting can be considerably trickier to implement than upcasting, and has less applicability.

```
function inject (request, fn, logger, imposterState) {  
  return new Promise((done, reject) => {  
    // Leave parameters for older interface  
    const injected = `(${fn})(config, injectState,  
logger, done, imposterState);`,  
  
    // ...  
  
    compatibility.downcastInjectionConfig(config);  
  
    // ...  
  });  
}
```

```
function downcastInjectionConfig (config) {  
  if (config.request.method || config.request.data) {  
    Object.keys(config.request).forEach(key => {  
      config[key] = config.request[key];  
    });  
  }  
}
```

Hidden interfaces

```
{  
  "is": { ... },  
  "_behaviors": [  
    "shellTransform": "node ./time.js"  
  ]  
}
```

time.js

```
var request = JSON.parse(process.argv[2]),  
    response = JSON.parse(process.argv[3]);
```



```
var request = JSON.parse(process.env.MB_REQUEST),  
    response = JSON.parse(process.env.MB_RESPONSE);
```

Obviousness



Elegance



Stability



Breaking hidden interfaces

In the worst case, hidden interfaces can break consumers using the published interface.

Without care, this will happen.

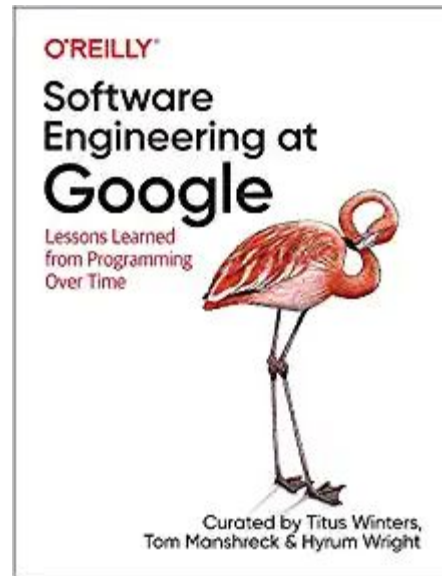
```
// Windows has a pretty low character limit to the command line. When we're in danger
// of the character limit, we'll remove the command line arguments under the assumption
// that backwards compatibility doesn't matter when it never would have worked to begin with
let fullCommand = `${command} ${quoteForShell(request)} ${quoteForShell(response)}`;
if (fullCommand.length >= maxShellCommandLength) {
  fullCommand = command;
}
```

How to think about API evolution

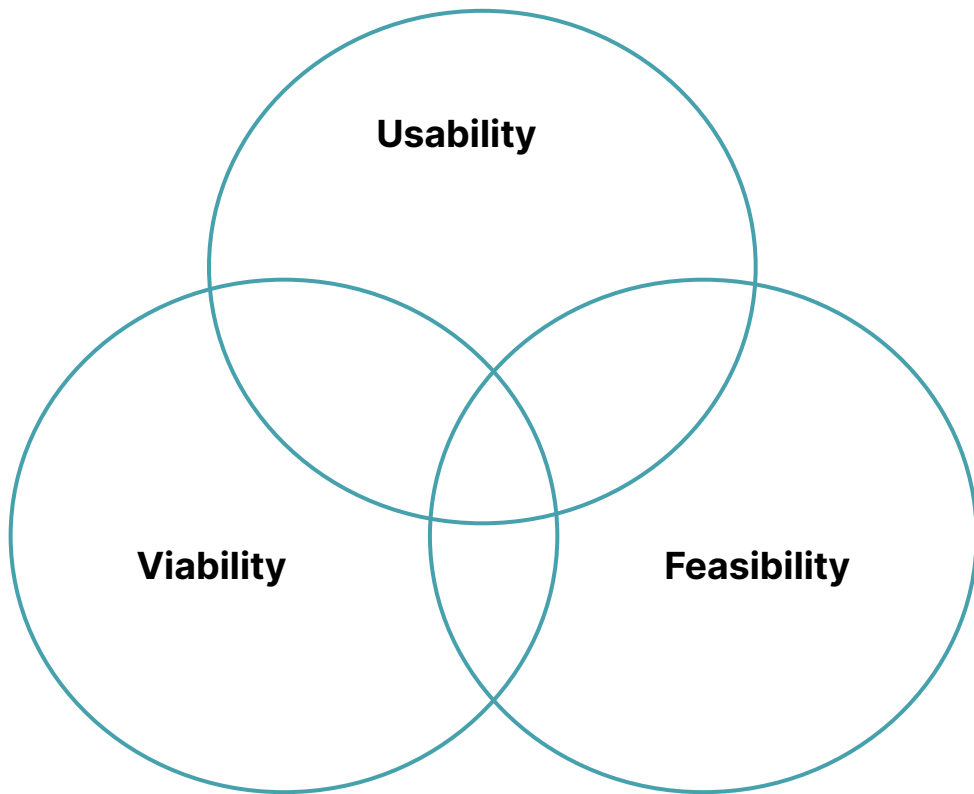
Hyrum's Law

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

www.hyrumslaw.com



API evolution is a product management concern



Product tradeoffs

Versioning is less an architectural solution than a product solution, and should be evaluated within a product framework.

Remember Hyrum's Law: your contract is a promise, not a guarantee!

Viability - are we solving a problem our users have?

Usability - are we making it easy to use our product?

Feasibility - can we do this in an architecturally sound way?

Reduction of cognitive load
or underlying complexity

Obviousness
Elegance
Stability

Protecting downstream
systems

Thank you!

Brandon Byars

Head of Technology

bbyars@thoughtworks.com