~~Homomorphic Event Sourcing~~

Consumer-Driven Contracts Done Right

symbiont
ARNAUD BAILLY

NICOLE RAUCH
software development &
development coaching

Overview

- Introduction
- Event Sourcing 101
- A Formal Model for Event Sourcing
- Consumer-Driven Contract Testing
- Beyond CDCT
- Conclusion

Part I

Introduction

Why?

or:

How this all began

Who?

Who we are?

What?

Formal Methods ∩ DDD

Languages, Type Systems,
Engineering Techniques

Part II

Event Sourcing 101

Ubiquitous Language

- Is . . . well . . . ubiquitous

Ubiquitous Language

- Is . . . well . . . ubiquitous
- Carries the business language into the code and beyond

Ubiquitous Language

- Is . . . well . . . ubiquitous
- Carries the business language into the code and beyond
- Allows everybody to understand everybody else

Ubiquitous Language

- Is . . . well . . . ubiquitous
- Carries the business language into the code and beyond
- Allows everybody to understand everybody else
- Understanding without (even unconscious) translation steps

Ubiquitous Language

- Is . . . well . . . ubiquitous
- Carries the business language into the code and beyond
- Allows everybody to understand everybody else
- Understanding without (even unconscious) translation steps
- Should be made explicit in a glossary

Commands and Events

- Commands represent interactions from the outside world

Commands and Events

- Commands represent interactions from the outside world
- They are requests to the application

Commands and Events

- Commands represent interactions from the outside world
- They are requests to the application
- Events are the system's replies

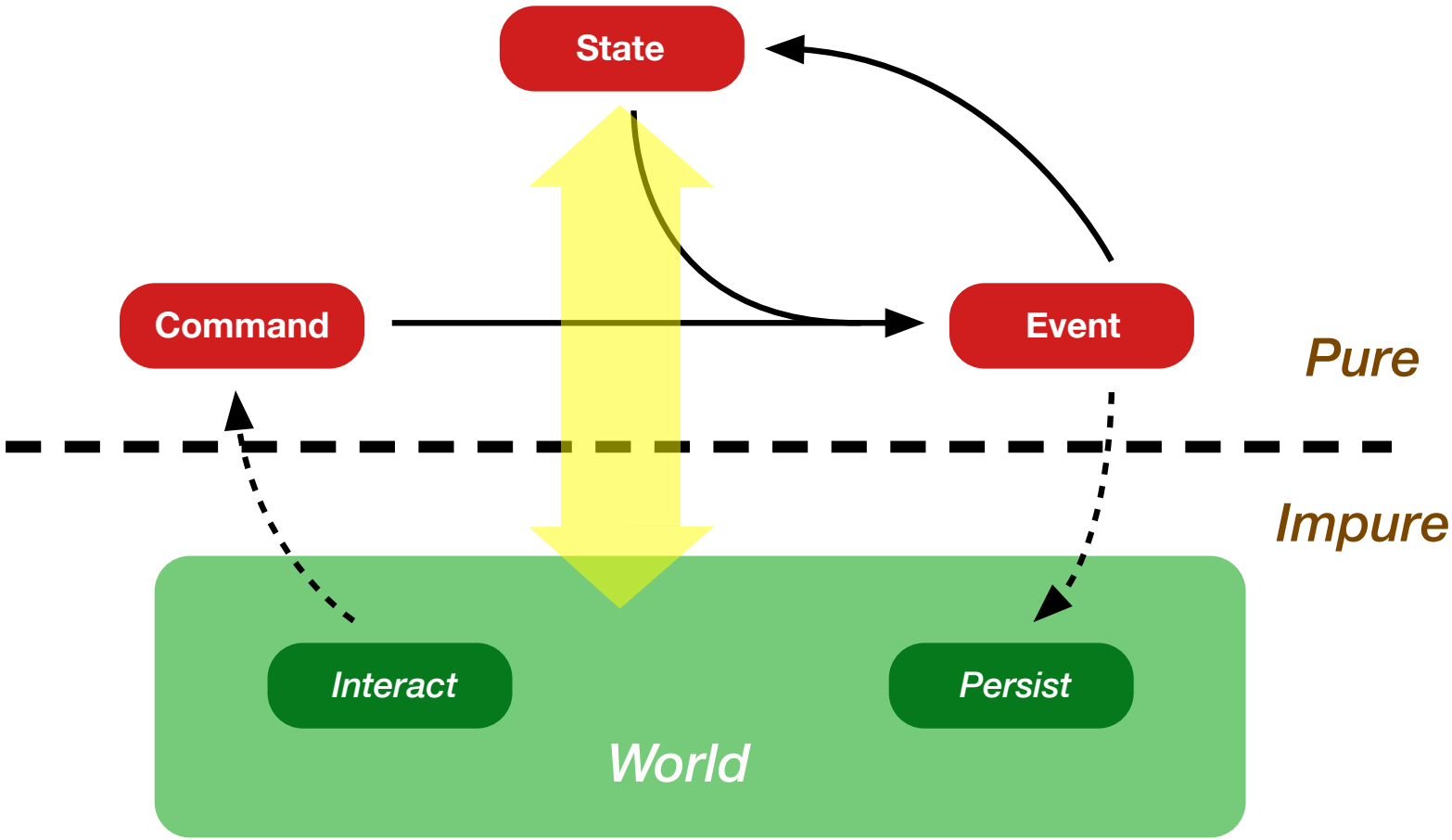
Commands and Events

- Commands represent interactions from the outside world
- They are requests to the application
- Events are the system's replies
- Events are persistently stored

Commands and Events

- Commands represent interactions from the outside world
- They are requests to the application
- Events are the system's replies
- Events are persistently stored
- The current state of the system is the result of all events that happened so far

Base Event Loop



Example: An Event-Sourced Pet Store

- Model (part of a) *Pets* online shop

Example: An Event-Sourced Pet Store

- Model (part of a) *Pets* online shop
- Owner can *Add* some pet or *Remove* it from the store

Example: An Event-Sourced Pet Store

- Model (part of a) *Pets* online shop
- Owner can *Add* some pet or *Remove* it from the store
- "Obvious" business rules: One cannot add the same pet twice or remove a non-existing pet

Inputs: Commands & Queries

```
data Input =
  -- Commands
    Add    { pet :: Pet }
  | Remove { pet :: Pet }
  -- Queries
  | ListPets
```

Outputs: Events & Answers

```
data Output =
  -- Events
    PetAdded   { pet :: Pet }
  | PetRemoved { pet :: Pet }
  -- Answers
  | Pets       { pets :: [ Pet ] }
  | Error      { reason :: PetStoreError }

-- some errors
data PetStoreError = PetAlreadyAdded
                   | PetDoesNotExist
```

Part III

A Formal Model for Event Sourcing

Event Sourcing as a Formal Language

- Conceptually, the commands and events comprise a so-called *alphabet*

Event Sourcing as a Formal Language

- Conceptually, the commands and events comprise a so-called *alphabet*
- A *word* is a valid sequence of letters from this alphabet

Event Sourcing as a Formal Language

- Conceptually, the commands and events comprise a so-called *alphabet*
- A *word* is a valid sequence of letters from this alphabet
- The set of all possible words is a *language*

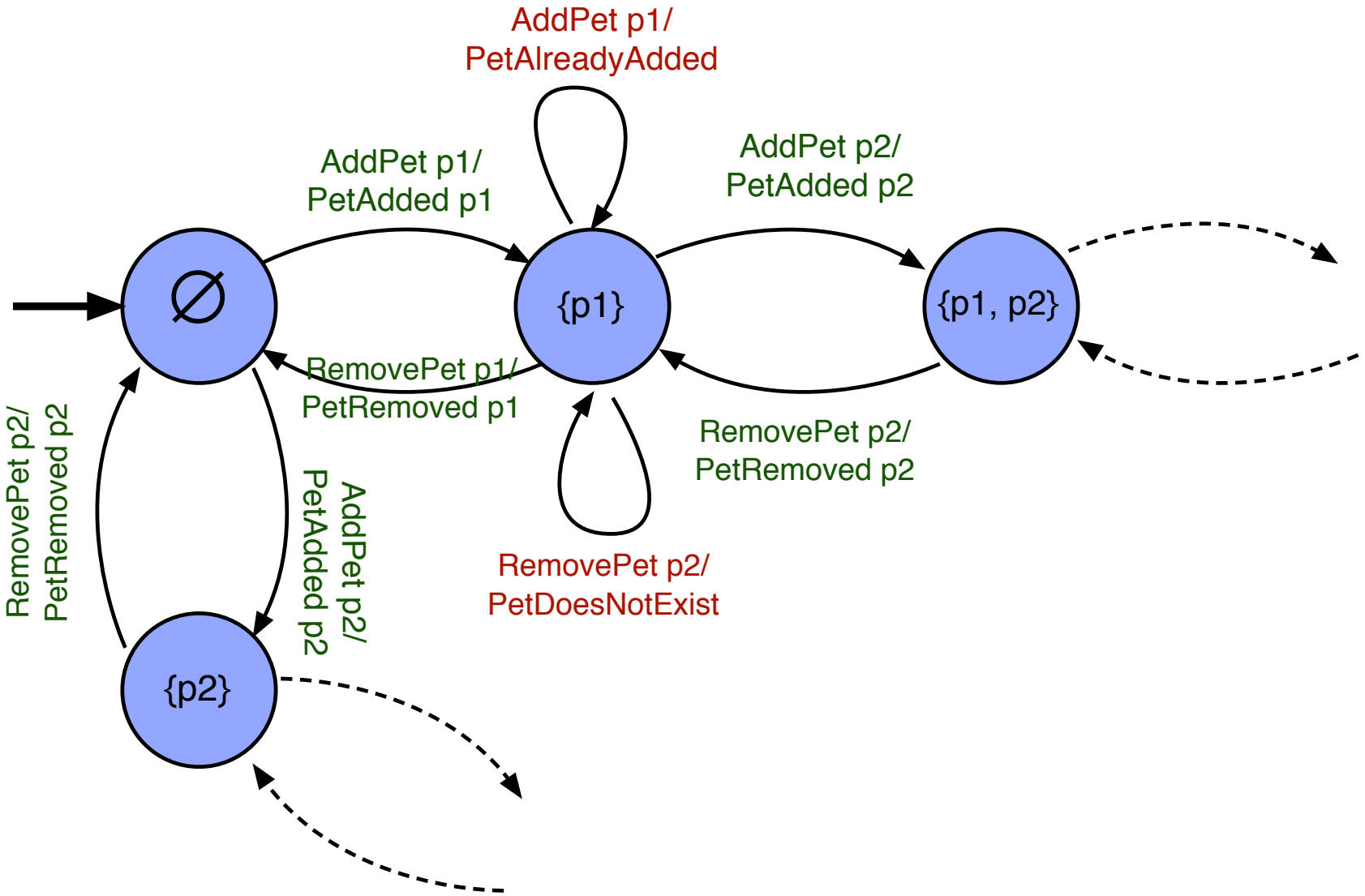
Event Sourcing as a Formal Language

- Conceptually, the commands and events comprise a so-called *alphabet*
- A *word* is a valid sequence of letters from this alphabet
- The set of all possible words is a *language*
- This means each word of the language corresponds to a *state* of the system

Event Sourcing as a Formal Language

- Conceptually, the commands and events comprise a so-called *alphabet*
- A *word* is a valid sequence of letters from this alphabet
- The set of all possible words is a *language*
- This means each word of the language corresponds to a *state* of the system
- So the language enumerates all the reachable states of the system

Example: Pet Store States



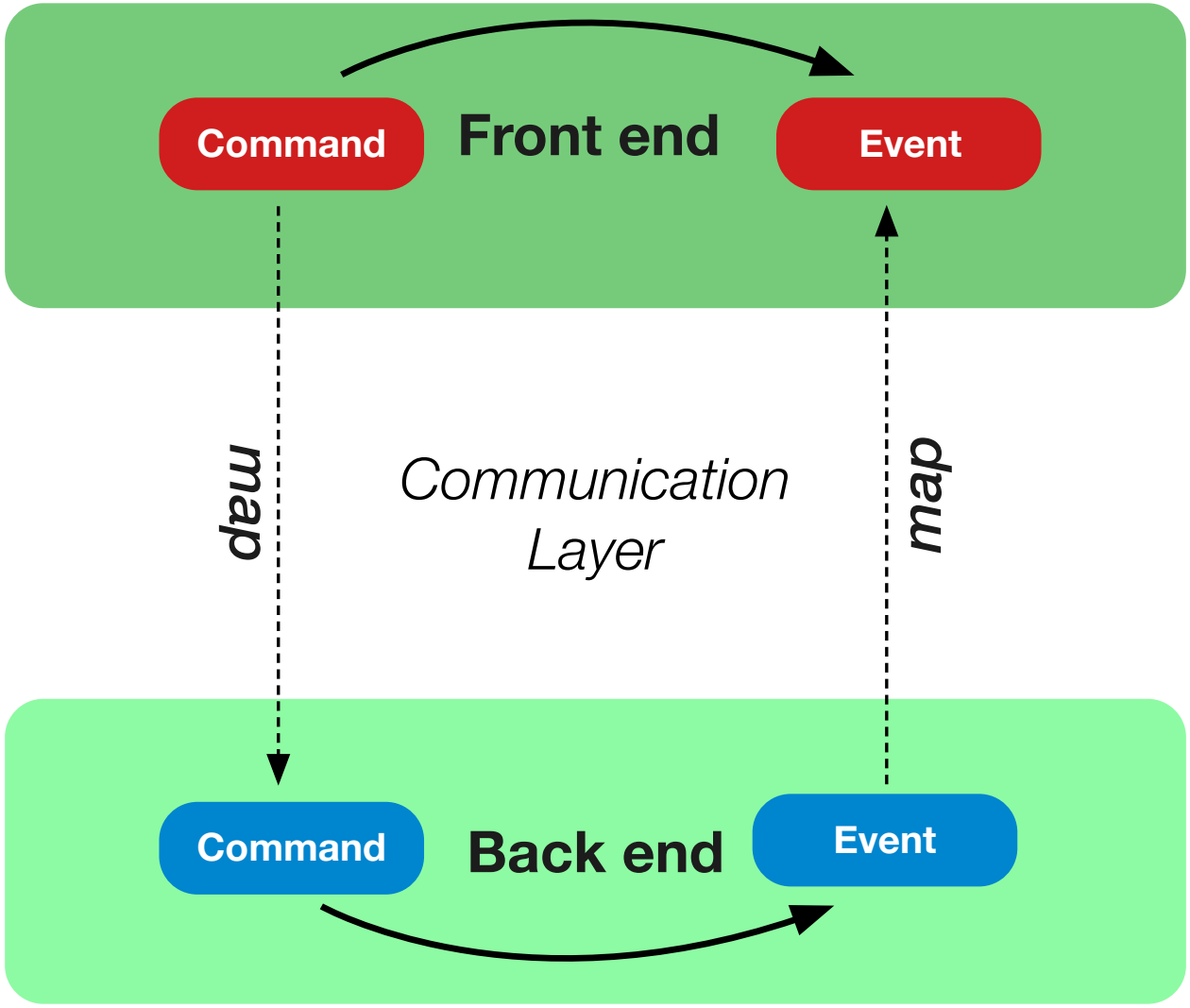
Part IV

Consumer-Driven Contract Testing

Standard Problem for Webapps?

- Testing that frontend and backend play nicely together

The Interaction Problem



Standard Approach: "Super-Naïve"

- Backend designs API
- Backend is developed, tests are written
- Frontend waits until backend is implemented
- Frontend is developed
- Tests for frontend are written
- Interaction is tested via integration tests

Standard Approach: "Super-Naïve"

Problems:

- Frontend development is blocked
- Integration tests are slow
- Full integration testing does not scale

Standard Approach: “Still Quite Naïve”

- Backend designs API
- Frontend writes mocks for backend API
- Backend is developed, tests are written
- Frontend can also be developed immediately
- Interaction tests for frontend use these mocks

Standard Approach: “Still Quite Naïve”

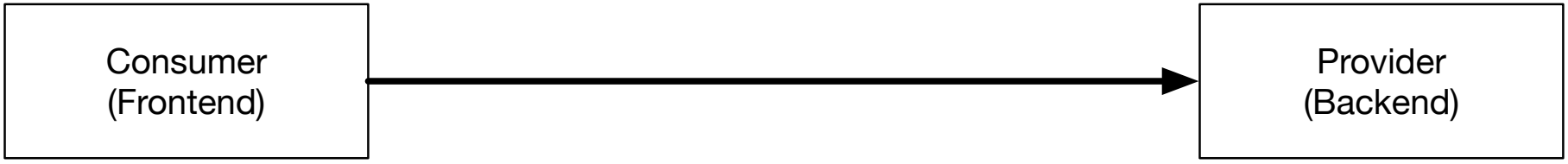
Problems:

- Frontend relies on mocks for backend API
- Do the mocks reflect the backend’s actual behaviour?
- Usually, backend behaviour changes
- Frontend does not notice because mocks still look good

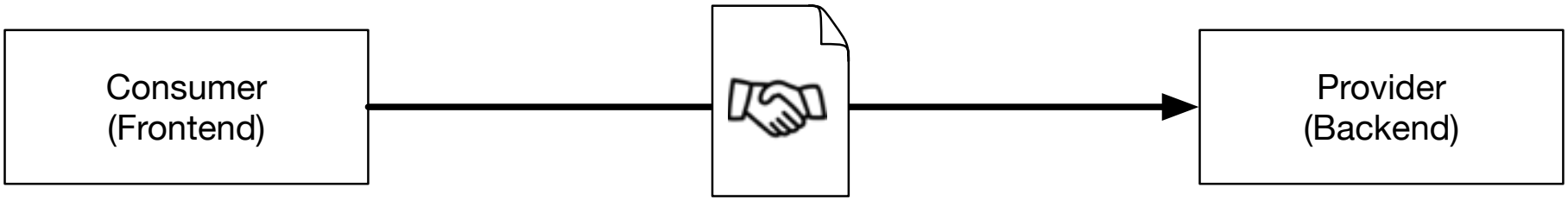
Standard “Industry-Strength” Approach

- Consumer-Driven Contract Testing
- Hand-written contracts

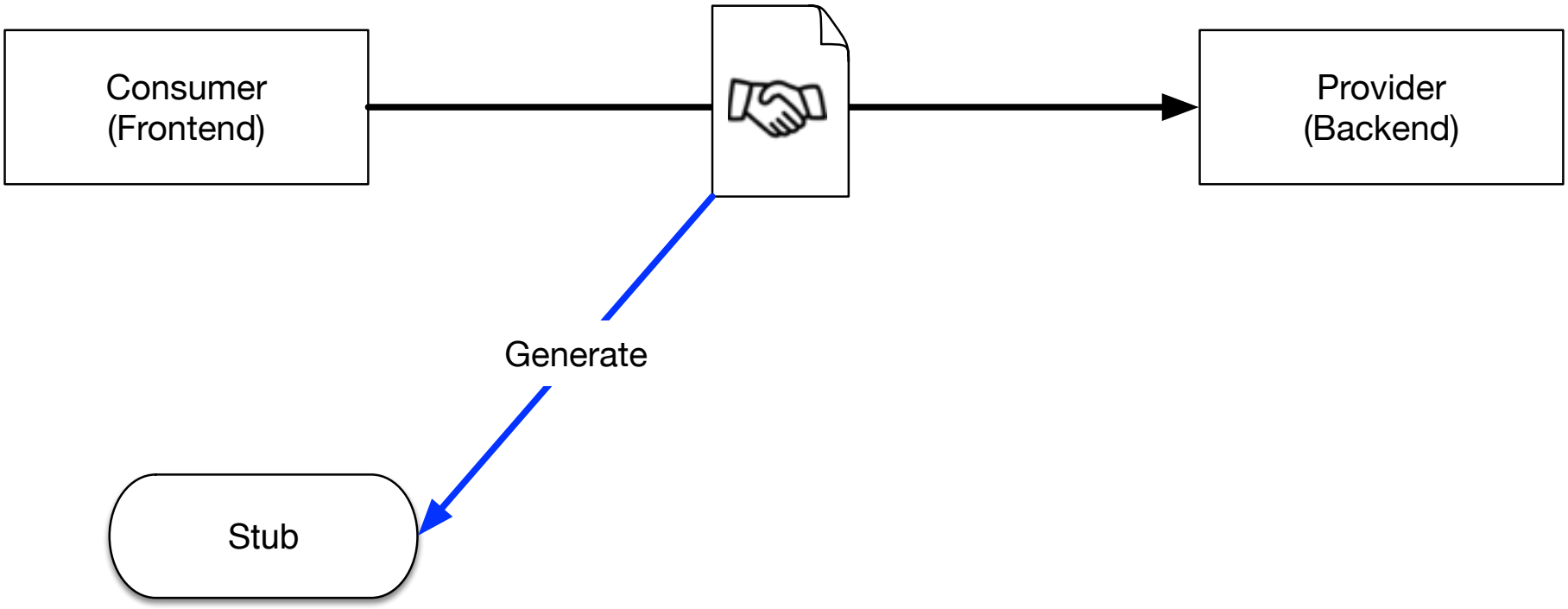
Standard “Industry-Strength” Approach



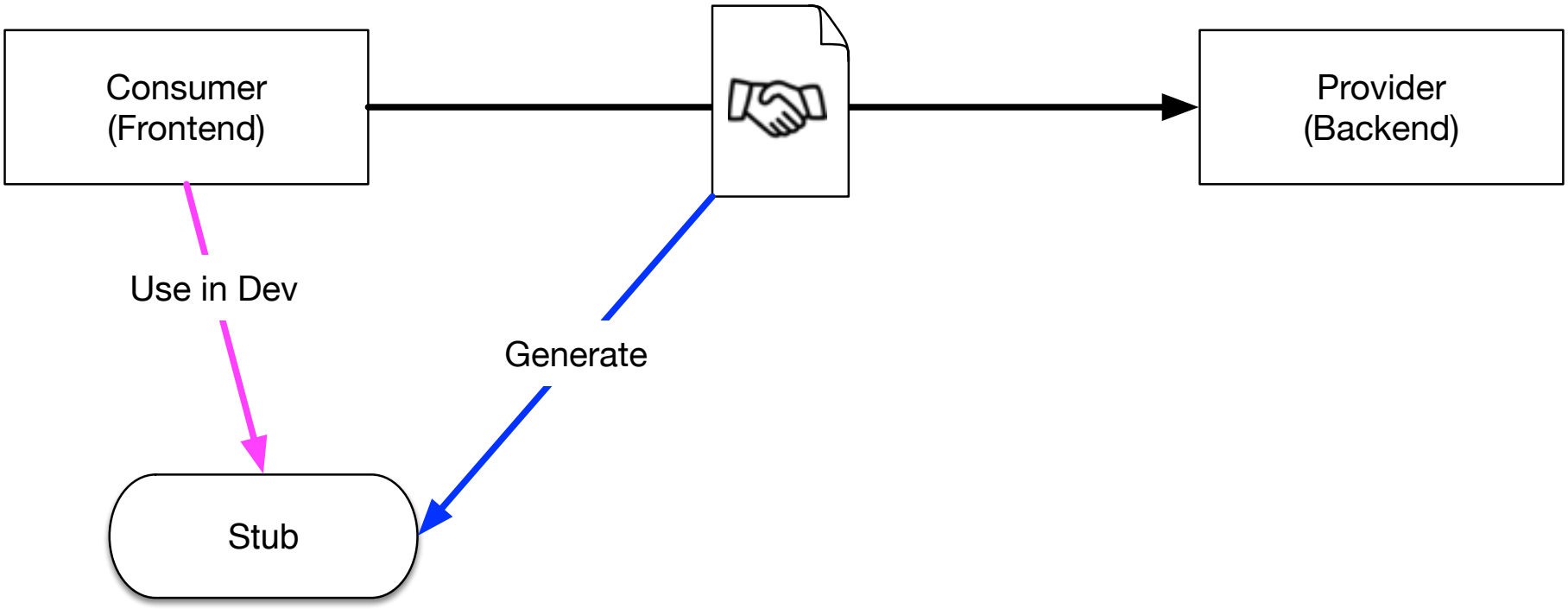
Standard “Industry-Strength” Approach



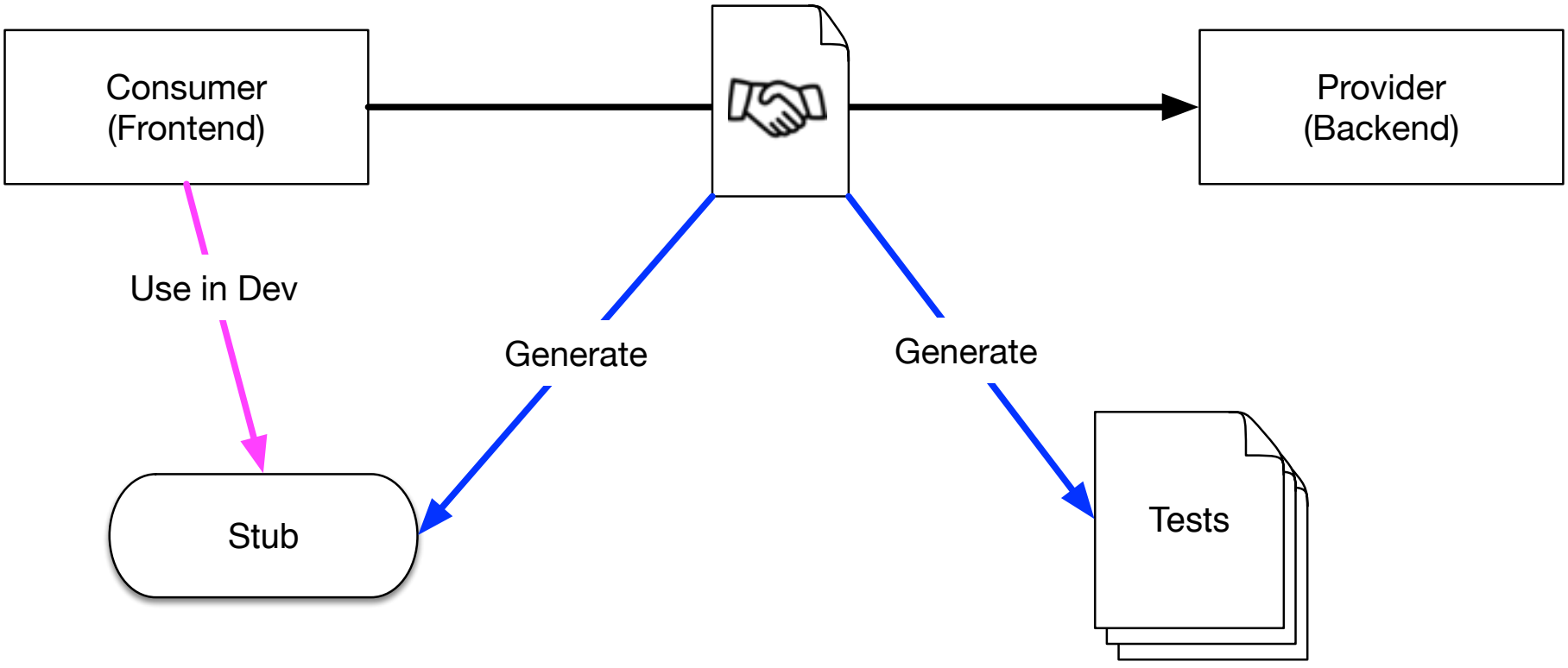
Standard “Industry-Strength” Approach



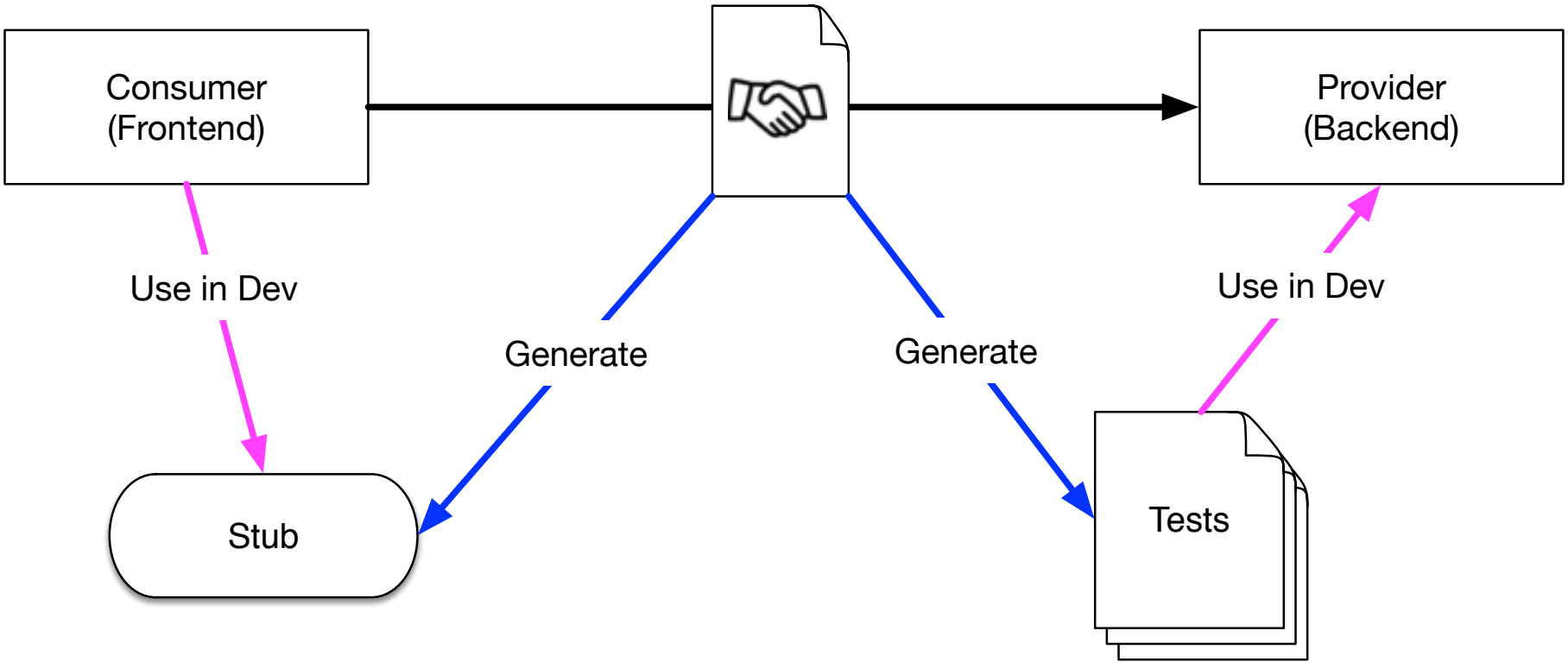
Standard “Industry-Strength” Approach



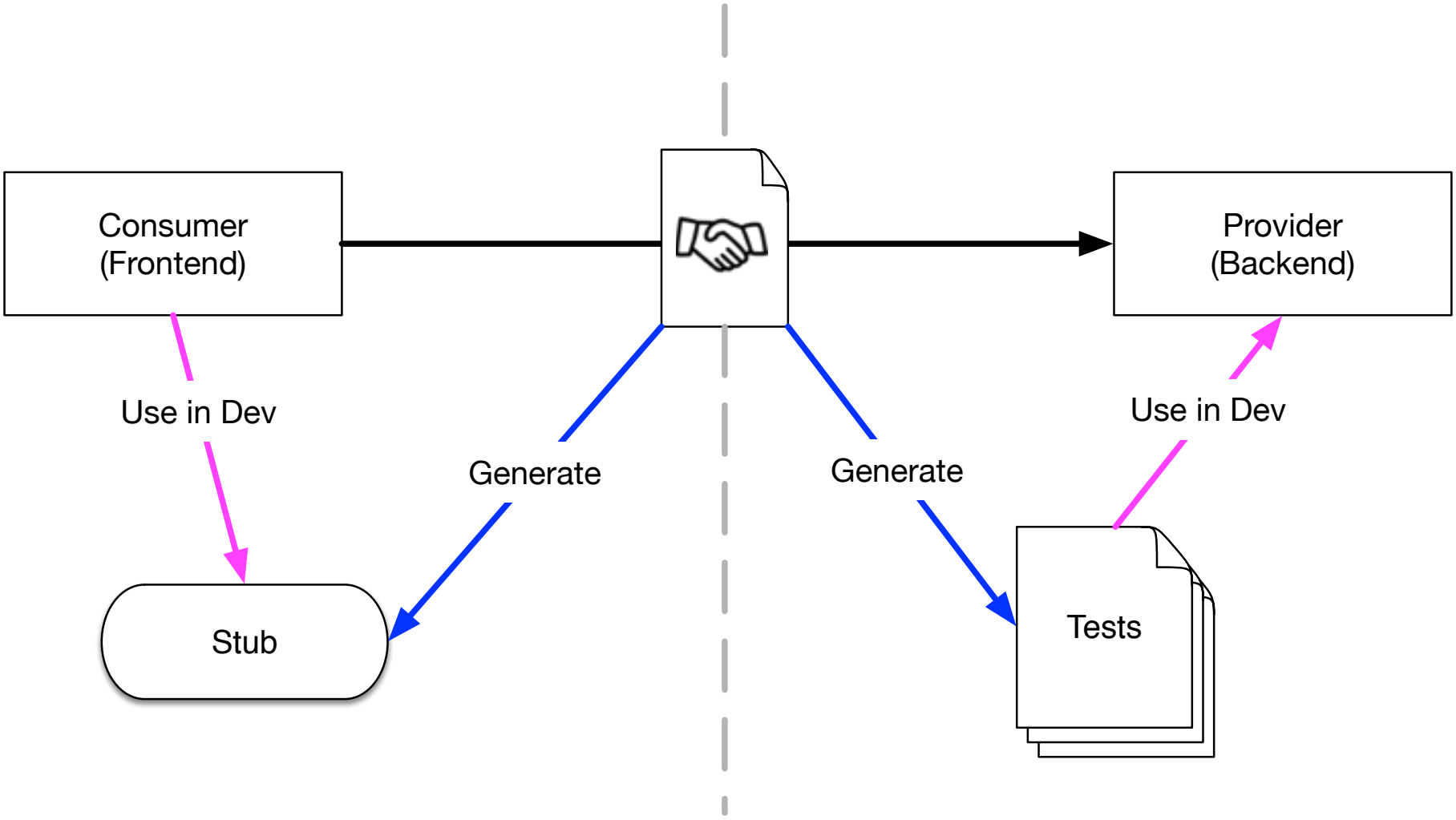
Standard “Industry-Strength” Approach



Standard “Industry-Strength” Approach



Standard “Industry-Strength” Approach



Example: Pet Store Contracts

```
{
  "description": "a request for all pets",
  "providerState": "i have no pets",
  "request": {
    "method": "GET",
    "path": "/pets",
    "headers": { "Accept": "application/json" }
  },
  "response": {
    "status": 200,
    "headers": { "Content-Type": "application/json" },
    "body": {
      "tag": "Pets",
      "pets": []
    }
  }
}
```

Example: Pet Store Contracts

```
{
  "description": "a request for all pets",
  "providerState": "i have a list of pets",
  "request": {
    "method": "GET",
    "path": "/pets",
    "headers": { "Accept": "application/json" }
  },
  "response": {
    "status": 200,
    "headers": { "Content-Type": "application/json" },
    "body": {
      "tag": "Pets",
      "pets": [
        { "petName": "Fifi",  "petType": "Dog" },
        { "petName": "Minki", "petType": "Cat" }
      ]
    }
  }
}
```

The Problem with CDCT

- Provider testing must manually establish the desired state

The Problem with CDCT

- Provider testing must manually establish the desired state
- Contract testing is only as good as its contracts

The Problem with CDCT

- Provider testing must manually establish the desired state
- Contract testing is only as good as its contracts
- Manual contract-writing can be tedious and even error-prone

The Problem with CDCT

- Provider testing must manually establish the desired state
- Contract testing is only as good as its contracts
- Manual contract-writing can be tedious and even error-prone
- Errors may only be discovered late in the process, when the backend implements some functionality and discovers that it does not match the contract

The Problem with CDCT

- Provider testing must manually establish the desired state
- Contract testing is only as good as its contracts
- Manual contract-writing can be tedious and even error-prone
- Errors may only be discovered late in the process, when the backend implements some functionality and discovers that it does not match the contract
- If contracts are too sparse, we miss out

The Problem with CDCT

- Provider testing must manually establish the desired state
- Contract testing is only as good as its contracts
- Manual contract-writing can be tedious and even error-prone
- Errors may only be discovered late in the process, when the backend implements some functionality and discovers that it does not match the contract
- If contracts are too sparse, we miss out
- If contracts are too verbose (or too many), testing takes too long

Part V

Beyond CDCT

Back to Formal Language

- Often contracts are fairly simple, just mapping requests to replies

Back to Formal Language

- Often contracts are fairly simple, just mapping requests to replies
- What if some request only makes sense in a certain state?

Back to Formal Language

- Often contracts are fairly simple, just mapping requests to replies
- What if some request only makes sense in a certain state?
- We need more information: Let's use *State Machines*!

Our “formal model” approach

- Describe the core domain interactions as a formally verifiable model

Our “formal model” approach

- Describe the core domain interactions as a formally verifiable model
- Generate mocks for the frontend: Use the State Machine as an *Acceptor*

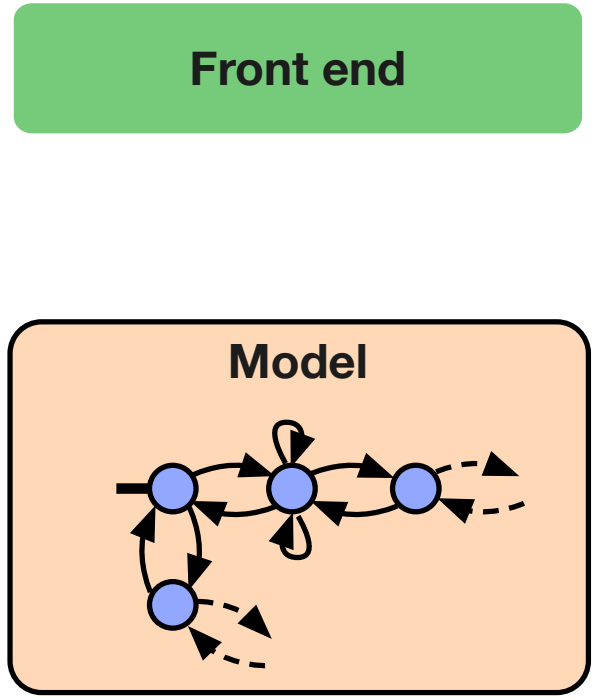
Our “formal model” approach

- Describe the core domain interactions as a formally verifiable model
- Generate mocks for the frontend: Use the State Machine as an *Acceptor*
- Generate tests for the backend: Use the State Machine as a *Generator*

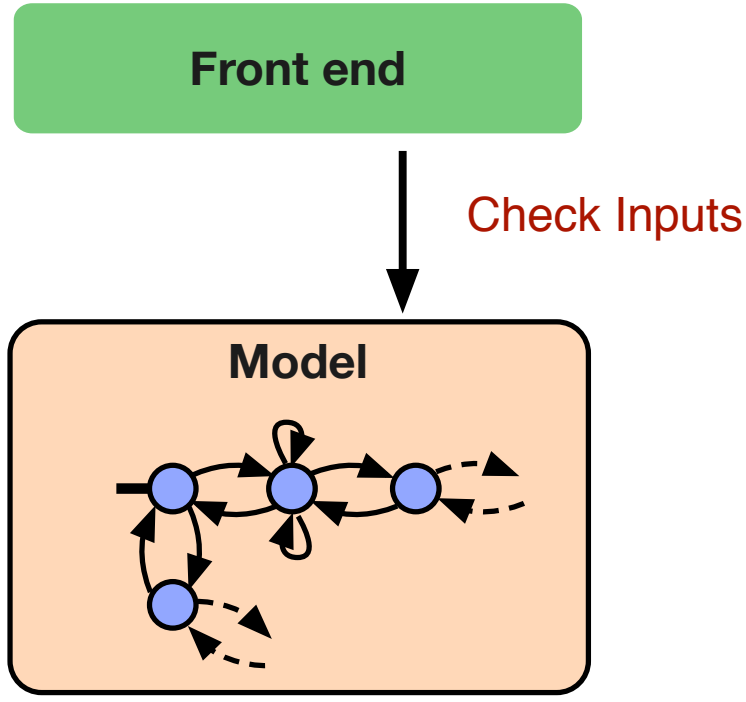
Our “formal model” approach

- Describe the core domain interactions as a formally verifiable model
- Generate mocks for the frontend: Use the State Machine as an *Acceptor*
- Generate tests for the backend: Use the State Machine as a *Generator*
- Guarantee: All aspects of the model are covered by tests and mocks

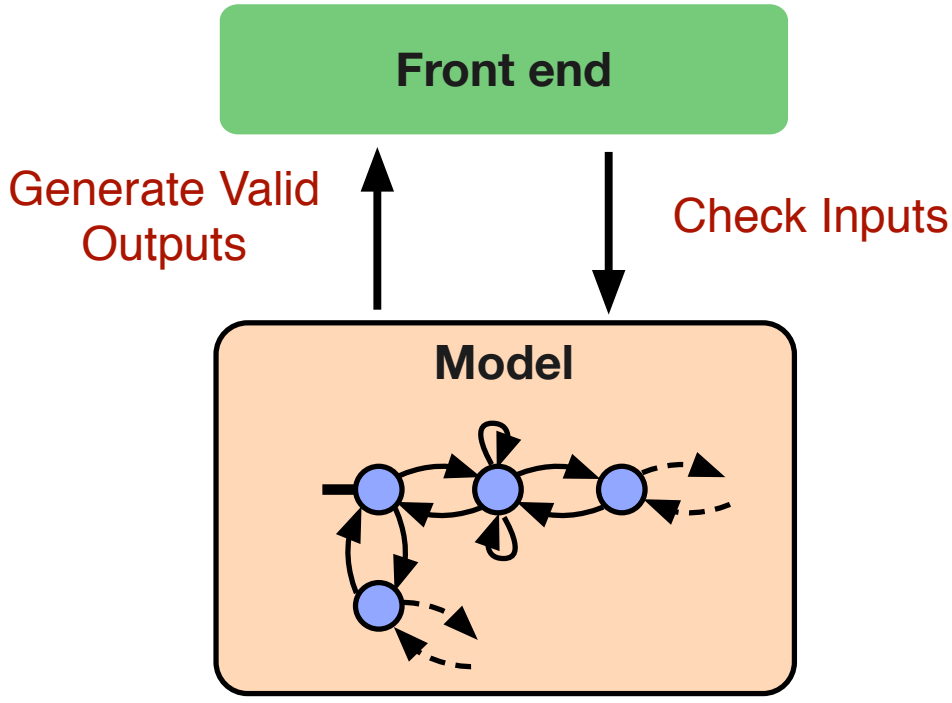
Model-Based Interaction Testing



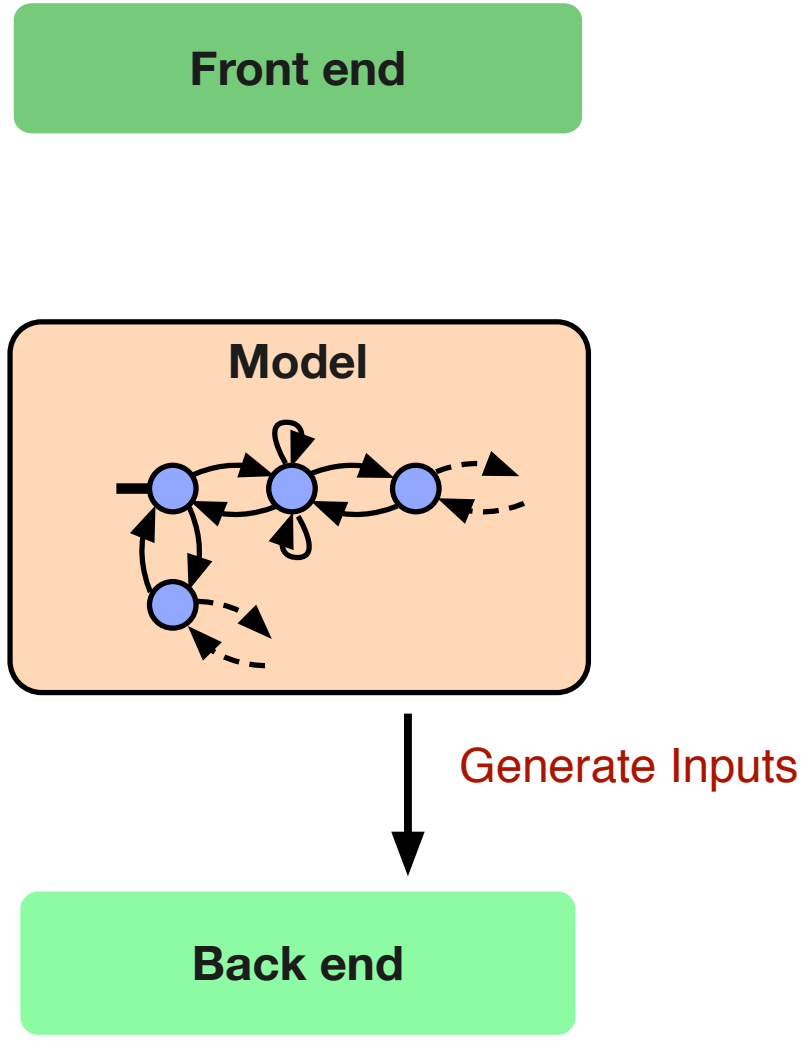
Model-Based Interaction Testing



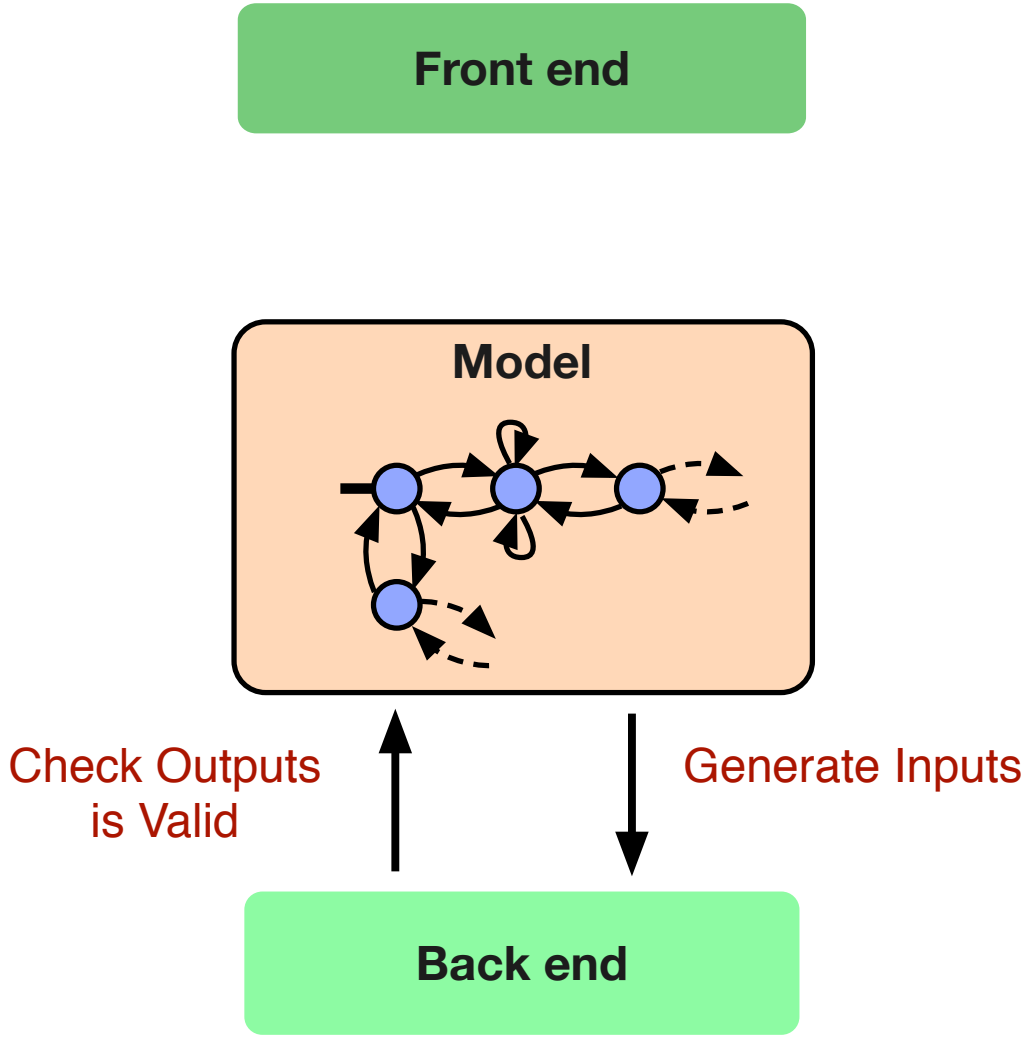
Model-Based Interaction Testing



Model-Based Interaction Testing



Model-Based Interaction Testing



Example: PetStore Model

```
petStore :: Input
         -> PetStore
         -> (Maybe Output, PetStore)

petStore Add{pet}  store@PetStore{storedPets}
  | pet `notElem` storedPets =
      (Just $ PetAdded pet,
       store { storedPets = pet:storedPets } )

  | otherwise                =
      (Just $ Error PetAlreadyAdded, store)
```

Demo

Validating the backend & frontend

Demo

Updating the model

The Code

https://github.com/aleryo/homomorphic-event-sourcing

Bug Trophy

- Incorrect routes definitions in the API leading to invalid queries

Bug Trophy

- Incorrect routes definitions in the API leading to invalid queries
- (*v2*) Forgetting to move basket's content back to store when user logs out

Bug Trophy

- Incorrect routes definitions in the API leading to invalid queries
- (*v2*) Forgetting to move basket's content back to store when user logs out
- (*v2*) Bad copy/pasting leading to `RemovePet` actually adding it

Real World Application

- Testing Implementation of a Smart Contracts transaction scheduling platform

Real World Application

- Testing Implementation of a Smart Contracts transaction scheduling platform
- Define a Model of the system in terms of *Actions*, observable *State* and potential *Failures* from components

Real World Application

- Testing Implementation of a Smart Contracts transaction scheduling platform
- Define a Model of the system in terms of *Actions*, observable *State* and potential *Failures* from components
- Generate sequence of *Action*

Real World Application

- Testing Implementation of a Smart Contracts transaction scheduling platform
- Define a Model of the system in terms of *Actions*, observable *State* and potential *Failures* from components
- Generate sequence of *Action*
- Run *Actions* in parallel against the Model and the Implementation

Real World Application

- Testing Implementation of a Smart Contracts transaction scheduling platform
- Define a Model of the system in terms of *Actions*, observable *State* and potential *Failures* from components
- Generate sequence of *Action*
- Run *Actions* in parallel against the Model and the Implementation
- Check reached states in Implementation is identical to the Model's

Part VI

Conclusion

Event Sourcing & Formal Methods

- Building an*Event Sourced* system yields opportunities to leverage more formal approaches to Verification & Validation

Event Sourcing & Formal Methods

- Building an*Event Sourced* system yields opportunities to leverage more formal approaches to Verification & Validation
- Modelling as a *State Machine* over a *Formal Language* seems a promising approach

Event Sourcing & Formal Methods

- Building an*Event Sourced* system yields opportunities to leverage more formal approaches to Verification & Validation
- Modelling as a *State Machine* over a *Formal Language* seems a promising approach
- Provides foundations to develop independent parts of the system and *validate* their interaction

Event Sourcing & Formal Methods

- Building an*Event Sourced* system yields opportunities to leverage more formal approaches to Verification & Validation
- Modelling as a *State Machine* over a *Formal Language* seems a promising approach
- Provides foundations to develop independent parts of the system and *validate* their interaction
- It takes time and energy to devise and refine a model!

Takeaways

Plans are worthless,
but planning is everything

Dwight D. Eisenhower

Takeaways

Models are worthless,
but modelling is everything

Nicole & Arnaud

Thank you very much!

Arnaud Bailly

E-Mail arnaud@aleryo.com

Twitter @dr_c0d3

Web http://aleryo.com

Web http://symbiont.io

Nicole Rauch

E-Mail info@nicole-rauch.de

Twitter @NicoleRauch

Web http://www.nicole-rauch.de