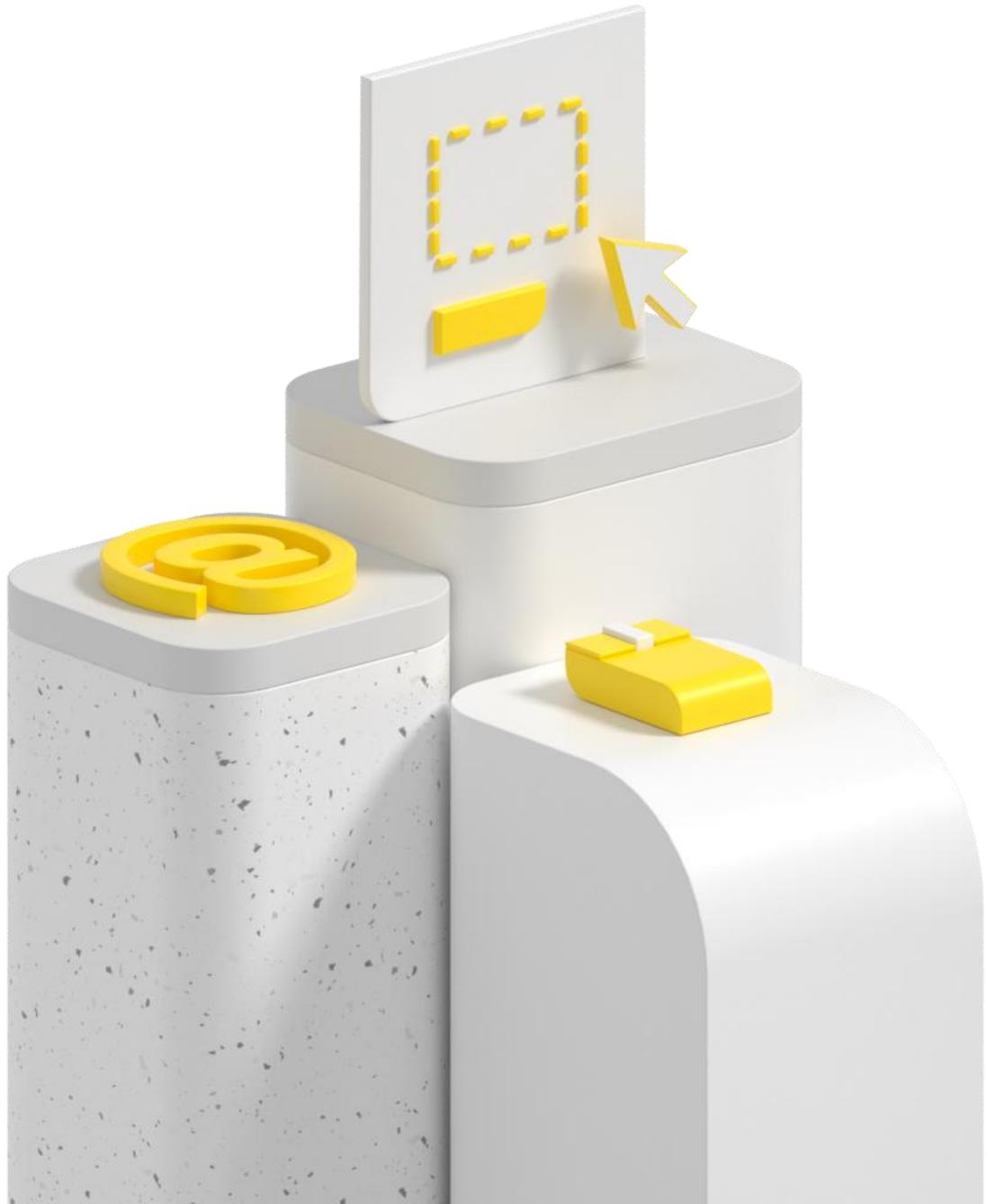
IT'S TINKOFF

Mockingbird

Или как убить всех зайцев одним выстрелом



Инеева Ольга: Тинькофф
@Olyainej

Содержание

- С какими проблемами тестировщики сталкиваются при тестировании интеграций
- Моки – что это, плюсы и минусы
- Mockingbird: что умеет?
- Эмуляция HTTP сервисов
- Эмуляция шинных сервисов
- Как мы используем mockingbird для ручных и автотестов

Команда

TCB – Tinkoff Credit Broker

Бизнес-линия Tinkoff Credit Broker
Продукты связаны с BNPL-политикой (buy now, pay later), например: автокредит, ипотека, сервис «Долями», потребительские кредиты и другое.

Много интеграций, как внутрибанковских, так и внешних, на стабильность работы которых влиять мы не можем по тем или иным причинам.

Боли тестирования интеграций

Еще не реализована

Работает нестабильно

Долго отвечает

Сложно получить определенный ответ

Необходимость «ручных» действий

Нехватка тестовых контуров

Решение - моки

Мок – эмуляция сервиса, с которым нам нужно работать.

Pros and cons

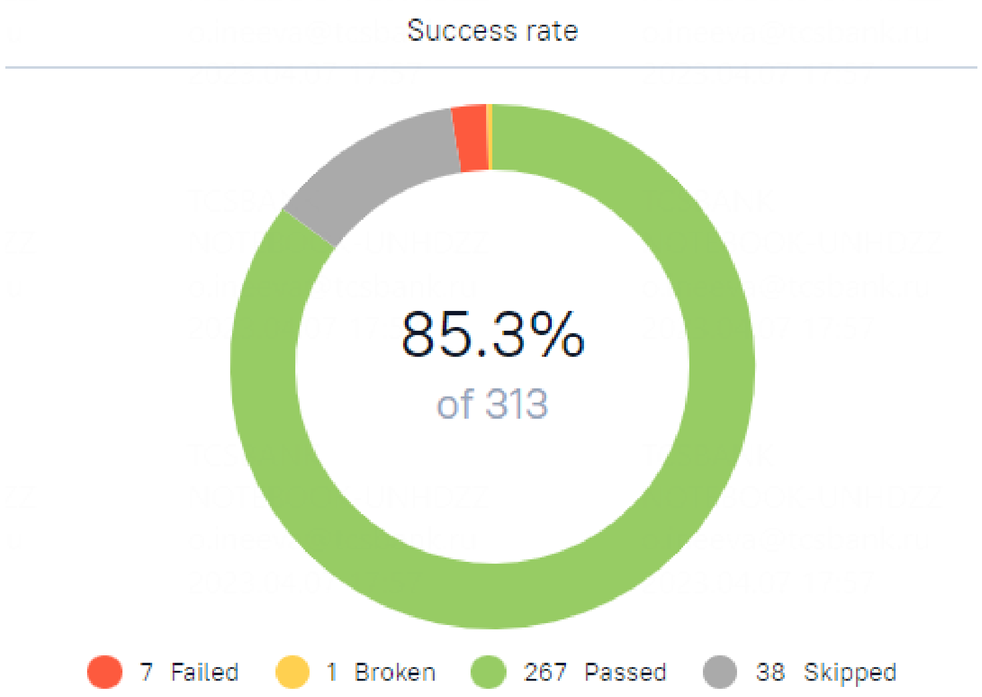
Плюсы

- Тесты становятся быстрее
- Можно наиграть все тестовые случаи
- Растет уровень доверия к тестам

Минусы

- Риск словить баг на реальной интеграции
- Нужно поддерживать в актуальном состоянии

Уровень доверия к тестам



Почему упал тест: не отвечает интеграция? Сломался сам тест?

Если упали тесты на моках – значит дело в нашей имплементации

Существующие популярные решения

Mountebank

Rest

Soap

Grpc (Плагин)

WebSocket (Плагин)

Wire Mock

Rest

Soap

Webhooks

Мокирование цепочки вызовов

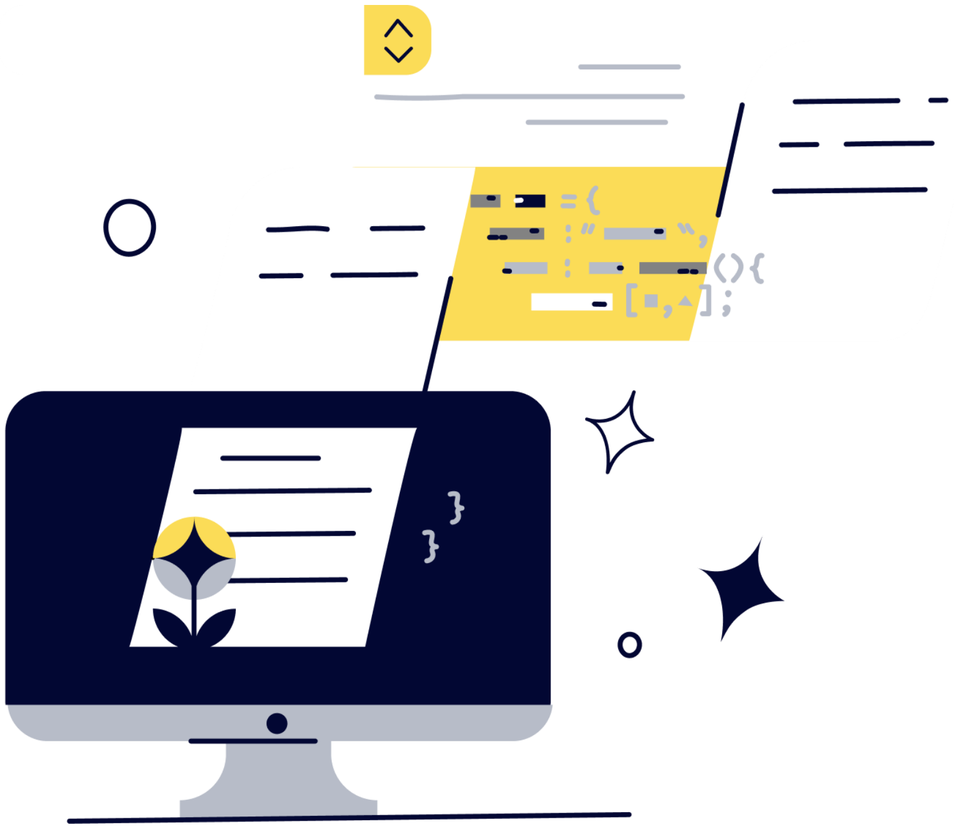
Чего не хватило?

- Нет эмуляции работы с брокерами сообщений
- Хотелось иметь и постоянный контур для Happy Path, и возможность тестировать конкретные сценарии, чтобы все работало через 1 инструмент
- Поддерживать сложные связанные сценарии, в т.ч. между HTTP моками и моками с брокерами сообщений

Mockingbird

Mockingbird – open-source проект, разработанный Даниилом Смирновым (Тинькофф).
Написан на Scala.

Возможности



- Эмуляция HTTP сервисов
- Эмуляция шинных сервисов
- Эмуляция GRPC сервисов

Типы конфигураций

- Тестовый контур – 1 шт
- Нужны заглушки, реагирующие по-разному при запросе на один и тот же url
- Не хочется каждый раз менять одну заглушку

3 типа конфигураций заглушек (по времени жизни и приоритету).

Конфигурации. Persistent

- Наименьший приоритет
- Не удаляются автоматически

Обеспечение автономности тестового контура

Конфигурации. Ephemeral

- Средний приоритет
- Удаляются каждую неделю

Временное изменение постоянного (persistent) мока

Конфигурации.

Countdown

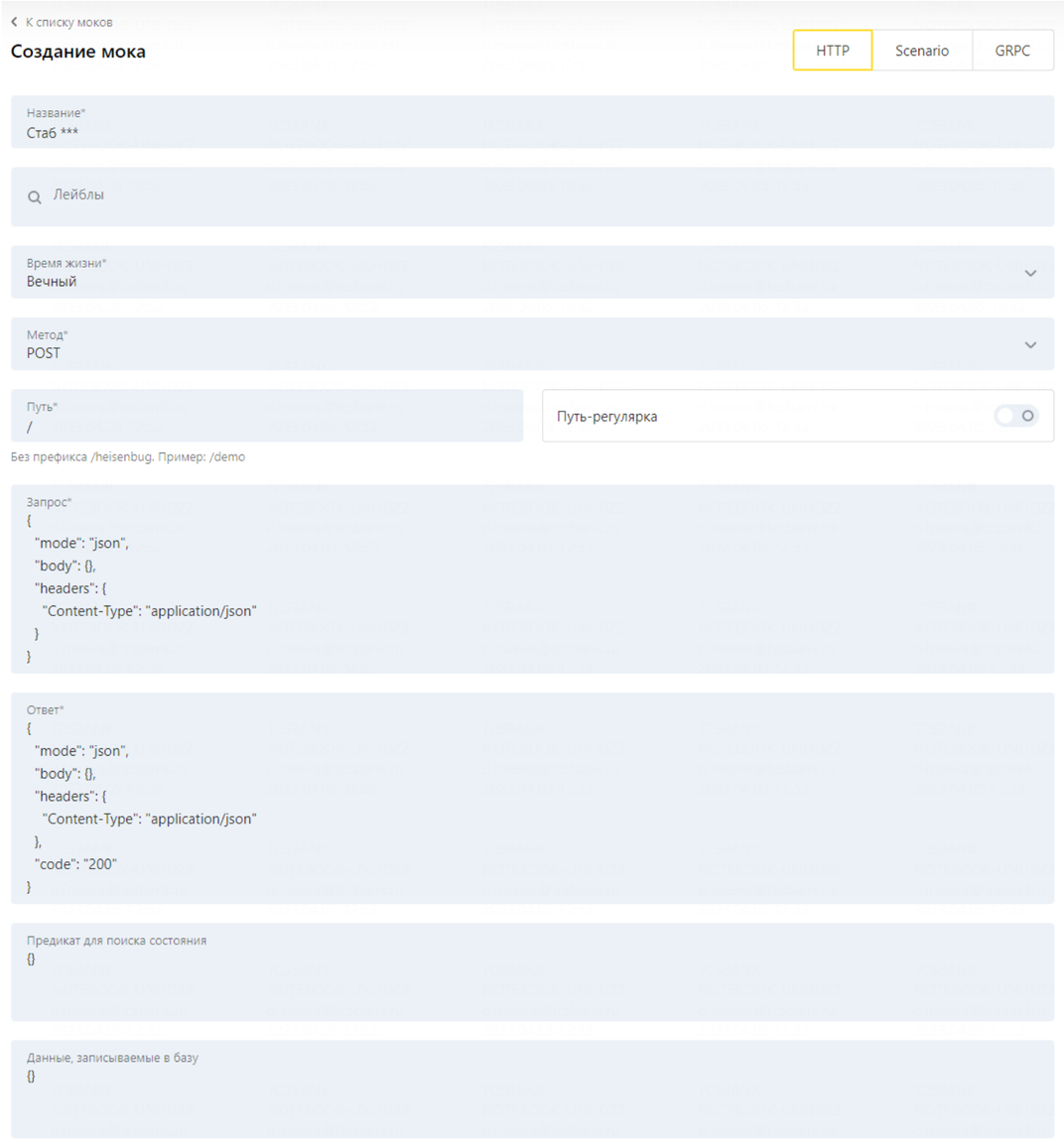
- Наивысший приоритет
- N раз
- Удаляются каждую ночь

Под конкретный сценарий

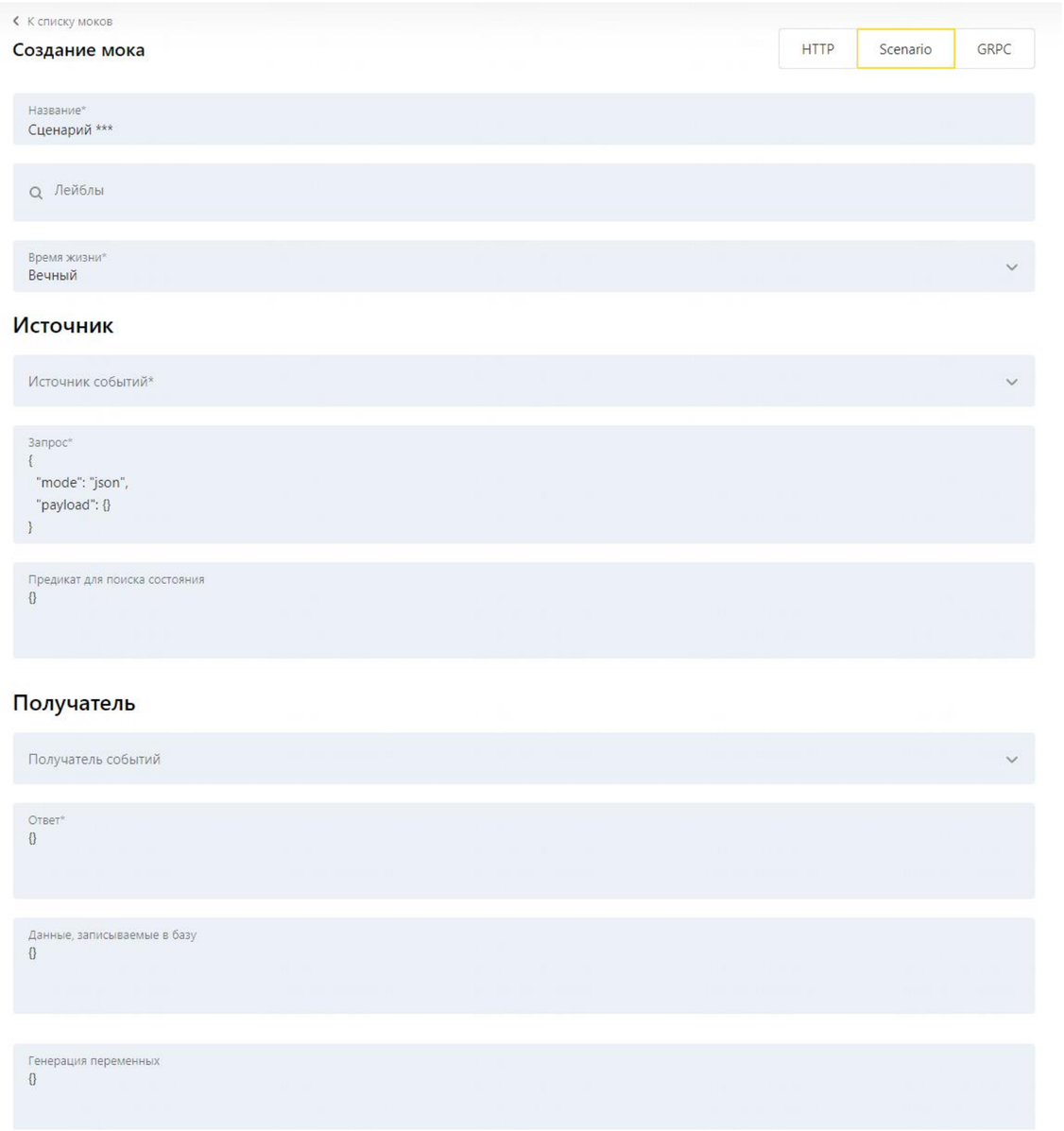
Как создать мок?

- Через API – для автотестов / с помощью Postman
- Через UI

UI. Создание HTTP заглушки



UI. Создание сценария для MQ



UI. Полезные фичи

Коллбэки

HTTP POST http://localhost:8228/api/mockingbird/exec/realty/requestPerson (delay 1 second)

Показать как JSON

Чтобы ваш МОК наверняка работал, рекомендуем предварительно сохранить, если вы вносили правки


```
{
  "path": "/heisenbug/httpCallbackDemo",
  "pathPattern": null,
  "name": "HTTP callback stub",
  "labels": [],
  "method": "POST",
  "scope": "persistent",
  "request": {
    "headers": {},
    "query": {},
    "body": {
      "person": {
        "==": "Ivan"
      }
    },
    "mode": "jlens"
  },
  "response": {
    "code": 200,
    "headers": {
      "Content-Type": "application/xml"
    },
    "body": {
      "msg": "ok, got you!"
    },
    "delay": "1 second",
    "mode": "json"
  },
  "state": null,
  "persist": null,
  "seed": null,
  "callback": {
    "type": "http",
    "request": {
      "url": "http://localhost:8228/api/mockingbird/exec/realty/requestPerson",
      "method": "POST",
      "headers": {
        "Content-Type": "application/json"
      },
      "body": {
        "firstName": "Ivan",
        "lastName": "Callbackovich"
      },
      "mode": "json"
    },
    "delay": "1 second"
  }
}
```

Эмуляция REST-сервисов

Валидация тела запроса

JSON, XML (полное или частичное совпадение), запрос без тела, запрос с любым непустым телом.

Сравнение параметров запроса

равенство/неравенство, больше/меньше числа, regexp, длина и существование поля

Режимы ответа

raw, json, xml, binary, proxy, json-proxy, xml-proxy

REST-сервисы. Пример.

```
{ "name": "Stub trigger body",
  "method": "POST",
  "path": "/heisenbug/stubBody",
  "scope": "persistent",
  "request": {
     "headers": {"Content-Type": "application/json"},
     "mode": "jlens", ← режим проверки
     "body": {
         "id": {"==": 42} ← Заглушка сработает, если в теле запроса поле id имеет значение 42
  },
  "response": { ← какой ответ получим при срабатывании заглушки
      "code": 200,
      "mode": "json",
      "body": {
         “field": "Hello from body trigger mock! ",
     },
     "headers": {"Content-Type": "application/json"},
     "delay": "1 second" ← Можем конфигурировать скорость обработки
   }
}
```

REST-сервисы. Query параметры. Пример

```
{
  "scope": "countdown",
  "method": "GET",
  "name": "Stub trigger query",
  "path": "/heisenbug/queryTest",
  "request": {
    "headers": {},
    "mode": "no_body",
    "query": {
      "data": {
        "==": 1234   ← заглушка сработает только если отправим GET запрос на URL /test/someMethod?data=1234
}}},
  "response": {
    "code": 200,
    "mode": "json",
    "body":{"query":"${query}"},   ← заглушка вернет ответ {"query": {"data": 1234}}
    "headers": {
      "Content-Type": "application/json"
    },
    "delay": "0 second"
  }}
```

Proxy mock

Есть необходимость переключаться между реальной интеграцией и моком?

Заглушка с ответом от мока

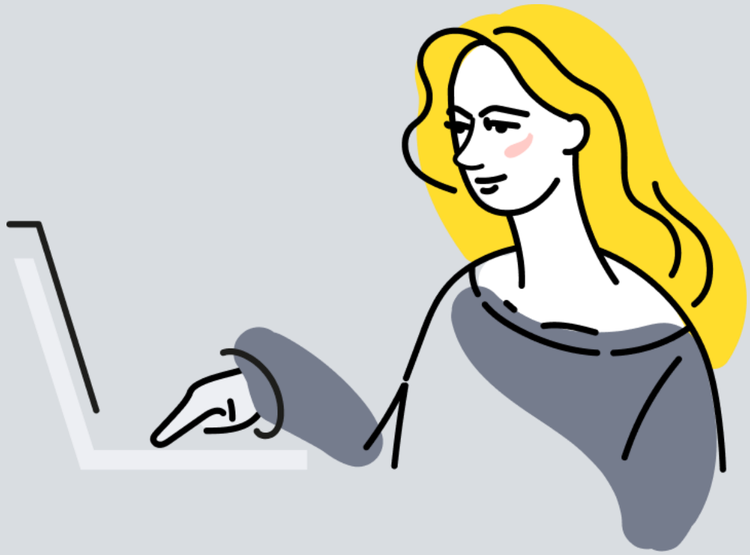
```
{
    "name" : "PROXY STUB COUNTDOWN",
   "scope" : "countdown", ← Имеет больший приоритет, выполнится первой
   "times" : 1,
   "method" : "POST",
   "path" : "/heisenbug/proxyDemo", ← Один и тот же урл
   "request" : {
      "body" : {},
      "mode" : "jlens",
      "headers" : {}
   },
   "response" : {
      "code": 200,
      "body" : {"message“ : "Response from mock"}, ← Получаем ответ от мока
      "delay" : "0 seconds",
      "mode" : "json"
   }
}
```

→

Заглушка с походом в реальный сервис

```
{
   "name" : "PROXY STUB"
   "scope" : “persistent", ← Имеет наименьший приоритет, выполнится в последнюю очередь
   "method" : "POST",
   "path" : "/heisenbug/proxyDemo", ← Один и тот же урл
   "request" : {
      "body" : {},
      "mode" : "jlens",
      "headers" : {}
   },
   "response" : {
      "mode" : "proxy" ← Указываем режим proxy
      "uri" : "http://google.ru", ← Ходим по этому урлу
      "delay" : "0 seconds",
    }
}
```

Эмуляция шинных сервисов



Mockingbird взаимодействует с брокерами сообщений через HTTP API, благодаря чему теоретически поддерживаются любые возможные MQ.

На практике: RabbitMQ, IBM MQ, Kafka

Эмуляция шинных сервисов. Режимы

input

- raw
- jlens
- json
- xml
- xpath

output

- raw
- json
- xml

Шинные сервисы. Пример

```
{"name": "Пришла весна",
 "source": "in_queue", ← конфигурация очереди источника
 "input": {
  "mode": "jlens"  // как для HTTP заглушек
  "payload": {
    "innerData.abc": { "==": “test"}} // как body для HTTP заглушек
 },
"destination": "out_queue", ← конфигурация очереди - пункта назначения
 "output": {
    "mode": “json",
    "payload": {
      "xyz": "abc"} ← что отправляем в очередь назначения
    },
    "delay": "1 second“
 }}
```

Callback

Нужно сходить по http куда-нибудь, получить ответ, а затем сходить еще куда-то/отправить сообщение в очередь?

С этим нам поможет механизм **callback**

HTTP callback. Пример

```
{// объявление заглушки...
"request": {
   "headers": {},
   "query": {},
   "body": {"person": {"==": "Ivan"}},
   "mode": "jlens“
},
"response": {
   "code": 200,
   "headers": {"Content-Type": "application/xml"},
   "body": {"msg" : "ok, got you!“},
   "delay": "1 second" ,
   "mode": “json“
},
"callback": { ← механизм callback
   "type": "http", ← тип callback - http
   "request": {
     "url": “/http://localhost:8228/api/mockingbird/exec/realty/requestPerson", ← Ходим по этому урлу
     "method": “POST",
     "headers": {"Content-Type": "application/json"},
     "body": {"firstName": "Ivan", "lastName": “Callbackovich"}, ← С этим телом
     "mode": "json"
   },
   "delay": "1 second",
   }
}
```

MQ callback. Пример

```
{// объявление заглушки...
"request": {
   "headers": {},
   "query": {},
   "body": {"person": {"==": "Ivan"}},
   "mode": "jlens“
},
"response": {
   "code": 200,
   "headers": {"Content-Type": "application/xml"},
   "body": {"msg" : "ok, got you!“},
   "delay": "1 second" ,
   "mode": “json“
},
"callback": { ← механизм callback
  "type": "message", ← тип callback - MQ
  "destination": "rmq_example_q1", ← конфигурация очереди
  "output": {
   "mode": "json",
   "payload": {"field": "value"} ← то, что отправляем в очередь
   }
  }
}
```

Генерация данных (Seeding)

Иногда возникает необходимость сгенерировать случайное значение и сохранить и/или вернуть его в результате работы мока.

Это можно сделать с помощью **seed**, положив туда сгенерированную строку/int/long/uuid/date/dateTime.

Генерация данных (seed). Пример

```
{// объявление свойств мока: url, scope, name...
  "seed": {
    "someId": "%{randomString(20)}, ← генерируем рандомную строку из 20 символов
    "someInt": "%{randomInt(10)}", ← генерируем рандомный Int в диапазоне от 0 до 10
    "someLong": "%{randomLong(30,40)}", ← генерируем рандомный Long в диапазоне от 30 до 40
    "someUUID": "%{UUID}", ← генерируем рандомный UUID
    "currentTime": "%{now(yyyy-MM-dd'T'HH:mm:ss)}", ← текущее время в заданном формате
    "currentDate": "%{today(yyyy-MM-dd)}", ← текущая дата в заданном формате
  "request": {
    "mode": "jlens",
    "body": {},
    "headers": {"Content-Type": "application/json"}},
  "response": {
    "mode": "json",
    "body": {
      "stringSeed": "${seed.someId}“ ←используем сгенерированное значение строки в ответе мока
      "intSeed": "${seed.someInt}", ←используем сгенерированное значение Int в ответе мока
      "longSeed": "${seed.someLong}", ←используем сгенерированное значение Long в ответе мока
      "uuidSeed": "${seed.someUUID}", ←используем сгенерированное значение UUID в ответе мока
      "timeSeed": "${seed.currentTime}", ←используем текущее время в заданном формате в ответе мока
      "dateSeed": "${seed.currentDate}"←используем текущую дату в заданном формате в ответе мока
    },
    "headers": {"Content-Type": "application/json"},
    "code": "200"},
  "state": null,
  "persist": null}
```

Состояния (State)

Иногда нужно сформировать более сложные сценарии, чем «отдать ответ A при теле запроса B поступающего на url C».

Например, нам нужно вызывать один и тот же метод по одному и тому же url-у, но реагировать по-разному

State. Пример. Заглушка 1

```
{
  "scope": "persistent",
  "path": "/heisenbug/checkStatus", ← Один и тот же URL здесь и далее
  "method": "POST",
  "name": "Статус готовности - Одобрено",
 "request": {
    "headers": {},
    "mode": "jlens",
    "body": {
      "test" : {"==": "test123"}}
  },
  "persist": {
    "_cardId":{"==":"${req.cardId}"}, "status": {"==":"approved"} ← Сохраняем в state cardId в поле _cardId и status = approved
  },
  "response": {
    "code": 200,
    "mode": "json",
    "headers": {"Content-Type": "application/json"},
    "body": {"message": "Карта находится в статусе: Одобрено"}  } ← Отправляем сообщение, что статус карты  - Одобрено
}
```

State. Пример. Заглушка 2

```
{
  "scope": "persistent",
  "path": "/heisenbug/checkStatus", ← Один и тот же URL
  "method": "POST",
  "name": "Статус готовности - Подтверждено",
 "request": {
    "headers": {},
    "mode": "jlens",
    "body": {
      "cardId" : {"~=": "test[a-zA-z0-9]+"}}
  },
  "state": {
    "_cardId":{"==":"${req.cardId}"}, "status": {"==":"approved"}}, ← Ищем наш state по cardId и status = approved
  "persist": {  "status": "confirmed"  } ← Обновляем state: status становится confirmed
  "response": {
    "code": 200,
    "mode": "json",
    "headers": {"Content-Type": "application/json"},
    "body": {"message": "Карта находится в статусе: Подтверждено"}  } ← Отправляем сообщение, что статус карты  - Подтверждено
}
```

State. Пример. Заглушка 3

```
{
  "scope": "persistent",
  "path": "/heisenbug/checkStatus", ← Один и тот же URL
  "method": "POST",
  "name": "Статус готовности - Выдано",
 "request": {
    "headers": {},
    "mode": "jlens",
    "body": {
      "cardId" : {"~=": "test[a-zA-z0-9]+"}}
  },
  "state": {
    "_cardId":{"==":"${req.cardId}"}, "status": {"==":"confirmed"}}, ← Ищем наш state по cardId и status = confirmed
  "persist": {  "status": "issued"  } ← Обновляем state: status становится issued
  "response": {
    "code": 200,
    "mode": "json",
    "headers": {"Content-Type": "application/json"},
    "body": {"message": "Карта находится в статусе: Выдано"}  } ← Отправляем сообщение, что статус карты  - Выдано
}
```

Использование в ручных тестах

- вечные http-моки
- вечные сценарии для MQ

полностью замокированный, изолированный тестовый контур, на котором можно наиграть любой тестовый сценарий, который нам необходим.
Бизнес-заказчик может пройти основной user-story фичи и проверить реализацию.

Использование в автотестах

- создание одноразовых http-моков
- создание одноразовых сценариев для MQ

мок/сценарий создается под конкретный автотест во время его прогона, если это требуется, используя API Mockingbird.

Использование в автотестах

```
private const val idReplaceString = "{{entityId}}"
private val contourReplaceString = "{{contour}}"

fun testScenario(
    entityId: String,
    contour: String?
): Response? {
    return MockingBirdClient.createScenario(
        readTemplate(“scenario.json")
            replace(idReplaceString, entityId)
            replace(contourReplaceString, contour)
    )
}
```

Создаем сценарий/заглушку

*createScenario вызывает API mockingbird для создания сценария.

Использование в автотестах

Имя сценария

Конфигурация

Количество использований

Какую очередь читаем

В какую очередь пишем

Условие срабатывания

Режим

Что пишем в очередь назначения

Не создаем и не обновляем state

Не генерируем рандомные данные

```
"name": "Test scenario {{entityId}}" ,
"labels": [],
"scope": "countdown",
"times": 1,
"source": "first_topic_{{contour}}",
"destination": "second_topic_{{contour}}",
"input": {
  "payload": {
    “someId": {
      "==": "{{entityId}}"}},
  "mode": "jlens“,
"output": {
    "payload": {
        "field": "value"
    },
    "delay": "0 seconds",
    "mode": "json"
  },
  "state": null,
  "persist": null,
  "seed": null
```

Mockingbird

- Решает проблему мокирования работы с шинными сервисами
- Обеспечивает наличие постоянного контура для Happy Path с возможностью тестировать частные случаи - с помощью 1 инструмента
- Предоставляет возможность писать сложные связанные сценарии

IT'S TINKOFF

Инеева Ольга

o.ineeva@tinkoff.ru @Olyainej