# Scenarigoの紹介と それを利用したメルペイでの インテグレーションテスト実例

APIテストツール４選！開発者が語る各ツールの特徴と魅力

30 November 2023

@zoncoen

# Who am I?

Kenta Mori

@zoncoen

Backend Engineer at Merpay, Inc.

# Agenda

- About Scenarigo
- Scenario Testing Platform in Merpay
- Conclusion

# About Scenarigo

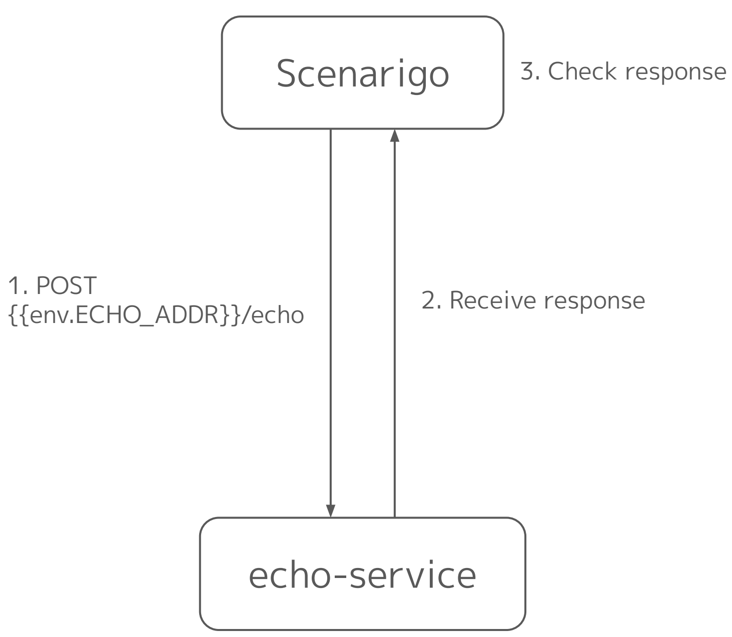
# Scenarigo

- WebアプリケーションのE2Eテストを行うためのツール
  - 2018年くらいから構想を始めて、ほそぼそと開発している
    - 業務で使えるものをつくろうと開発しはじめたけど半分趣味
    - https://github.com/zoncoen/scenarigo
  - 当時は以下のようなツールが（恐らく）なかったので自作することに
    - YAMLでテストシナリオを書ける
    - テストシナリオの使いまわしができる
    - HTTP/gRPCが使える
    - Goで拡張することができる

# Scenarigo

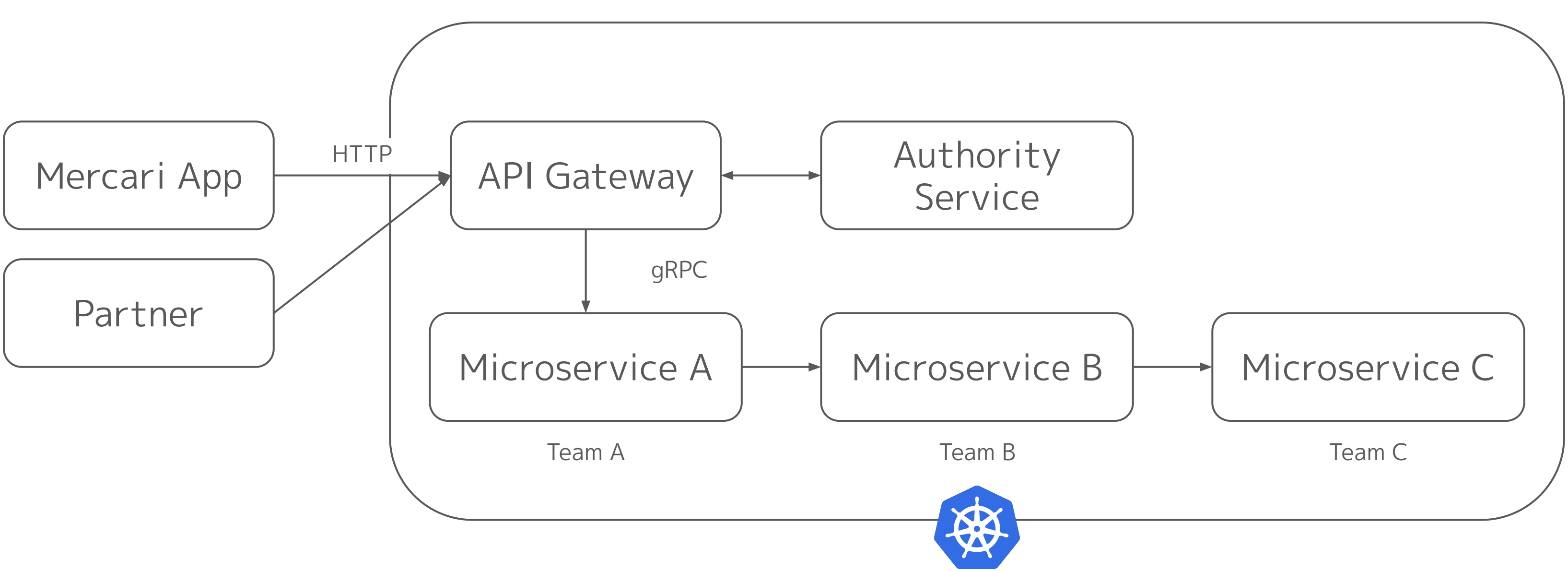
```
$ scenarigo run example.yaml
```

```yaml
title: echo-service
steps:
- title: POST /echo
  protocol: http
  request:
    method: POST
    url: '{{env.ECHO_ADDR}}/echo'
    body:
      message: hello
  expect:
    body:
      message: '{{request.message}}'
```



# Background

- k8sクラスタ上にマイクロサービスアーキテクチャで実装
  - 基本的に各マイクロサービスは異なるチームが開発している



# Motivation

- 各マイクロサービスでのインテグレーションテストが必要
  - 特にメルペイリリース前は複数のマイクロサービスの開発を一斉に始めたため、頻繁に開発環境が壊れていた
  - 定期的にテストする仕組みがなかったため、デグレに即座に気づきにくい
  - 依存しているマイクロサービスを無条件に信用せず、自分たちの開発しているマイクロサービスが正しく動作することに責任をもちたい
- QAチームがテストの自動化もすすめていたが、利用していたツールが自分たちのユースケースにマッチしていない部分もあった
  - QAチームだけでなくMSの開発者自身も簡単にテストができるようにしたい

**※ 2018年当時の話です**

# Goal

- YAMLでシナリオが定義できる
  - JSON等より比較的手で読み書きしやすい
    - GitHub上でdiffが見やすい
  - 直接コードを書くより記述量を少なくできる
- シナリオの使いまわしができる
  - ユーザーのログインなど共通の処理はコピペしたくない
- gRPCが使える
  - ほぼすべてのMSがgRPCサーバーなため
- Go のコードで拡張ができる
  - 拡張性の確保、社内のMSはほぼGoで開発されていたため
  - e.g. ある特定のルールでのID生成

# Include Example

```
title: echo
steps:
- title: login
  include: ./login.yaml < 読み込むシナリオファイルを指定する
  bind:
    vars:
      token: '{{vars.token}}' < login.yamlの結果をbind
- title: POST /echo
  protocol: http
  request:
    method: POST
    url: 'http://{{env.SERVER_ADDR}}/echo'
    header:
      Authorization: 'Bearer {{vars.token}}' < bindした値を利用する
    body:
      message: hello
  expect:
    code: 200
    body:
      message: '{{request.message}}'
```

# gRPC Example

```
title: Echo
plugins:
  grpc: grpc.so
steps:
- title: Echo
  protocol: grpc
  request:
    client: '{{plugins.grpc.CreateClient(ctx, env.TEST_ADDR)}}'
    ^ gRPCクライアントを渡す
    method: Echo < Method指定
    message:
      messageBody: hello
  expect:
    code: OK
    message:
      messageBody: '{{request.messageBody}}'
```

# Plugin

- Goで機能を拡張できる
  - Goのモジュールがそのまま使える
    - Goでサーバーを実装しているならロジックの再利用もできる
  - Go標準のプラグイン機能を利用
    - 便利な反面制約も多い
      - main moduleとpluginは同じGoのバージョンでコンパイルされていなければならない
      - すべての依存モジュールは同じバージョンでなくてはならない
      - Windowsでは動かない
    - ある程度いい感じにやってくれる scenarigo plugin build コマンドがある
      - 同じGoコンパイラを使う
      - 複数バージョンの依存がある場合は最新のバージョンで統一してビルドする

# Plugin

```
$ scenarigo plugin build
```

```yaml
schemaVersion: config/v1
scenarios:
- scenarios
pluginDirectory: ./gen
plugins:
  date.so:
    src: ./plugins/date
```

```go
package main

import "time"

func Today() string {
  return time.Now().Format(Layout)
}
```

```yaml
title: echo
plugins:
  date: date.so
steps:
- title: POST /echo
  protocol: http
  request:
    method: POST
    url: '{{env.ECHO_ADDR}}/echo'
    body:
      message: '{{plugins.date.Today()}}'
  expect:
    code: 200
```

# Why Use the std plugin?

- 普通のコードと同じ書き味で実装できるため
  - hashicorp/go-pluginのようにRPCでやるみたいな方法もあるが、それ用の書き方が必要になる
  - structなどをそのまま扱えるのでそのままメソッドを呼び出したりできて楽
- 利用者少ないからか新しいバージョンでデグレしたりするけど報告したらすぐ対応してもらえる
  - https://github.com/golang/go/issues/58920
  - https://github.com/golang/go/issues/62598
  - いつの間にか注意書きが追加されていた
- ~~使ってみたかったから~~

# Philosophy

- Scenarigoを開発する上で少し意識していること
  - できるだけシンプルにとどめる（利便性とのトレードオフではある）
    - 複雑な機能を使いこなすYAMLを正しく書くのは難しい
      - 複雑なことをやりたいときはGoで書いて普通のアプリケーションと同じようにテストも書いたほうがいい気がしている
  - 実行速度はあまり気にせず必要なければ高速化はしない
    - 実行時間はサーバー応答時間が支配的（であろう）
    - YAML <-> JSON <-> Go の変換を複数回やることもある
  - 書きやすさよりも読みやすさを重視する
    - 多少冗長になっても読んだときに明確なほうがメンテナンスしやすい
    - ソフトウェアにとってもメリットがあることもある

# Explicitly Define

```yaml
title: echo
steps:
- title: login
  include: ./login.yaml
  bind:
    vars:
      token: '{{vars.token}}'
- title: POST /echo
  protocol: http
  request:
    method: POST
    url: 'http://{{env.SERVER_ADDR}}/echo'
    header:
      Authorization: 'Bearer {{vars.token}}'
    body:
      message: hello
  expect:
    code: 200
    body:
      message: '{{request.message}}'
```

明示的にbindした値のみ以降のstepで参照できる
-> どこで定義されているかこのファイルを読むだけでわかる

LSPなどを実装するときにも楽（な気がしている）

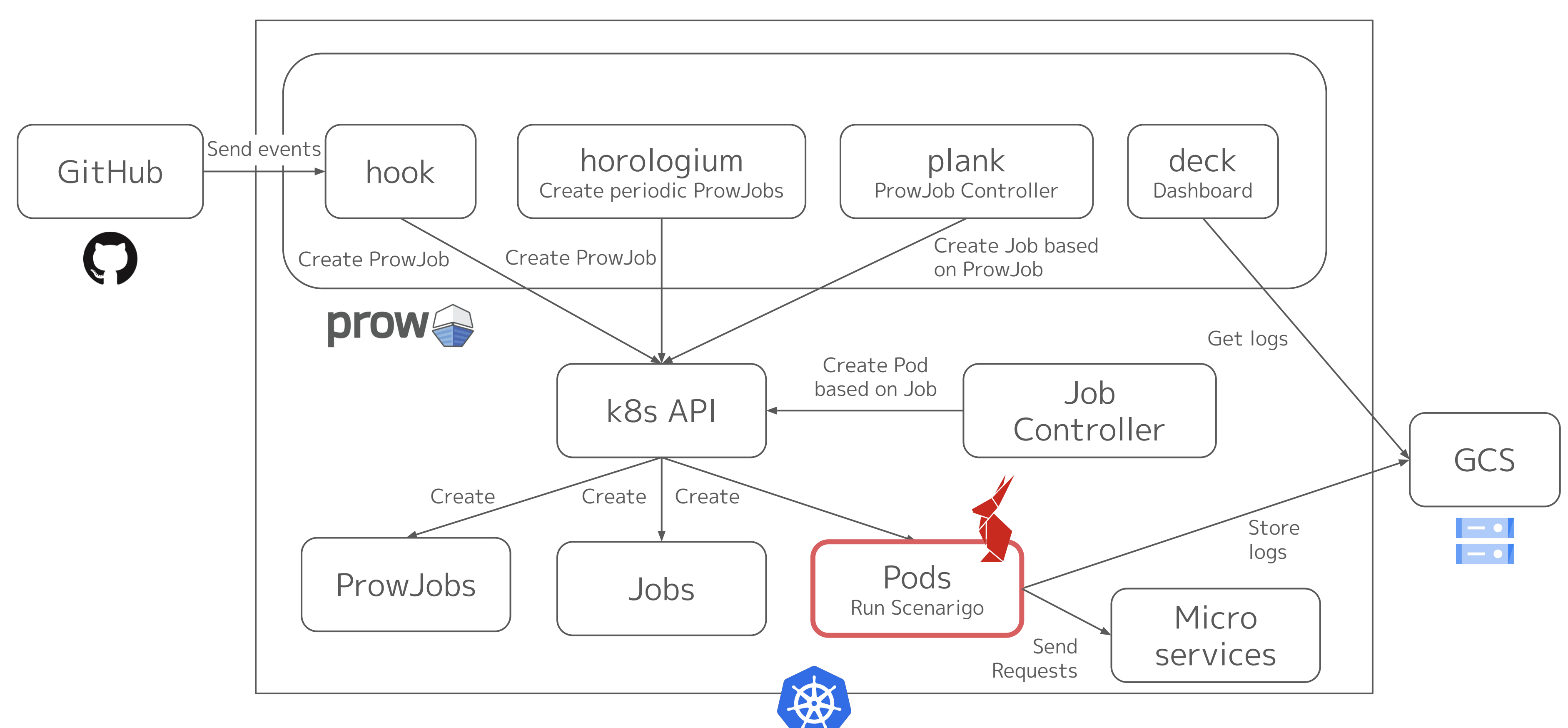
# Scenario Testing Platform in Merpay

# Customize

- Scenarigoはあくまでリクエストを投げてレスポンスを検証するだけのツール
  - 実際開発フローに組み込むには他にも必要なものがある
- 社内向けにScenario Testing Platformとして以下のようなものを提供している
  - CIの仕組み
  - CIと同じようにローカルから実行するためのラッパーコマンド
    - 共通処理をまとめたライブラリ、プラグインを含む
    - Goで処理を拡張できる点が役に立っている
  - PRのコードをビルド/デプロイして、自動でそのPodを使ってテストする機能
    - 今回は省略

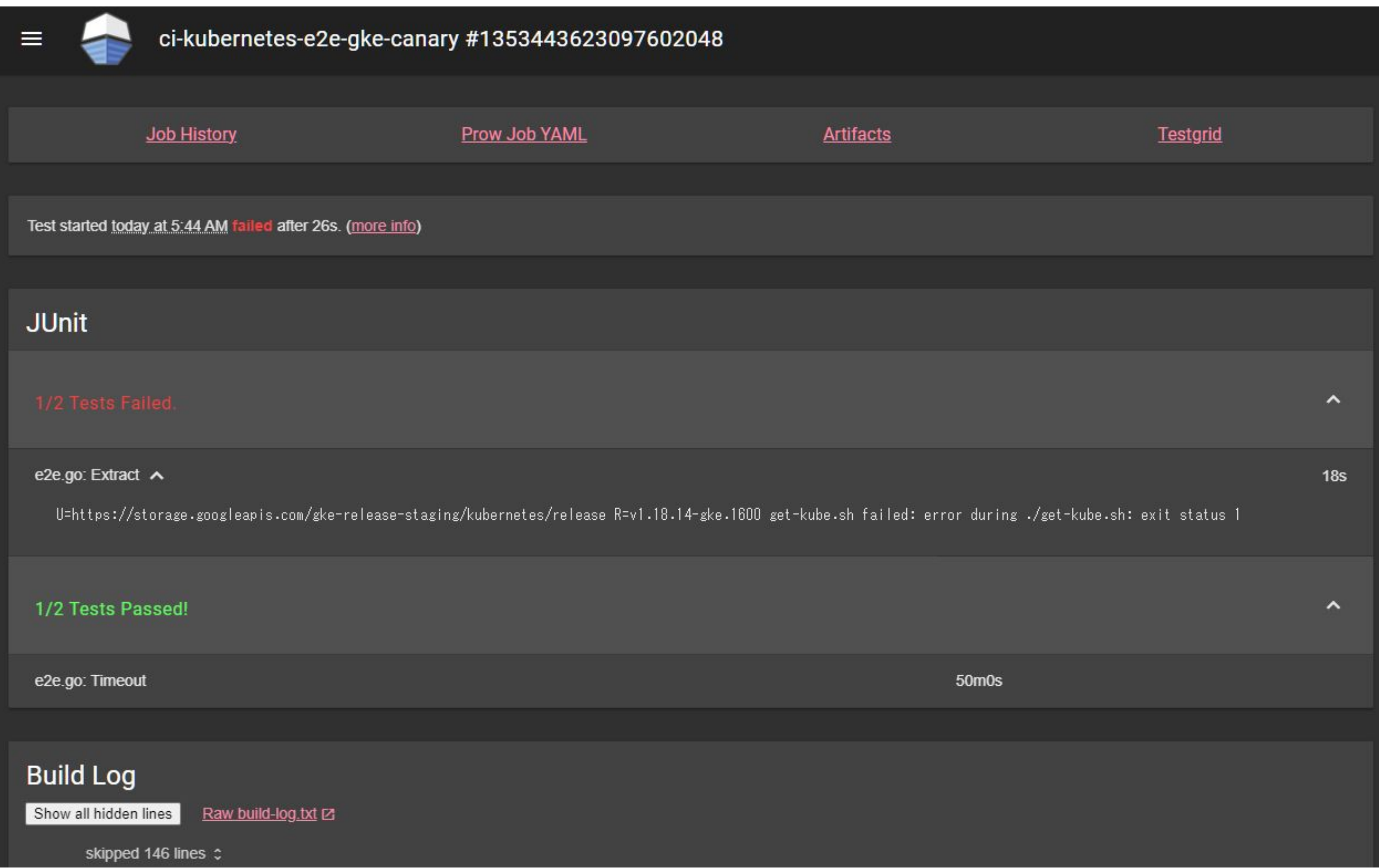
# CI

- Prow
  - k8s上で動くCI/CDシステム
    - https://github.com/kubernetes/test-infra/tree/master/prow
  - GitHubのPRをtriggerにテストを実行したりできる
  - Merpayはすべての MSが同じクラスタ上にいるので、そのクラスタ上でテストを実行すれば一般的なMS間通信と同じようにリクエストを投げられるので便利


GitHub
Send events
hook
horologium
Create periodic ProwJobs
plank
ProwJob Controller
deck
Dashboard
Create ProwJob
Create ProwJob
Create Job based
on ProwJob
prow
Get logs
Create Pod
based on Job
k8s API
Job
Controller
GCS
Create
Create
Create
Store
logs
ProwJobs
Jobs
Pods
Run Scenarigo
Micro
services
Send
Requests

# Dashboard

# Result



# Run Test on Local

- テストシナリオを作成するときには手元からテストを実行したい
  - 開発環境の各マイクロサービスはexposeされてないためインターネット越しに直接叩くことはできない
  - 各自接続できるように設定するのは大変
- Envoyを使った専用のReverseProxyを用意
  - Host ヘッダーを見てルーティング
  - 接続先の変更やヘッダーの付与はラッパーコマンドと共通ライブラリが自動で行うため、ユーザーは何も意識せずともCIでもローカルでも同じようにテストを実行できる
  - IP制限をかけているのでVPNにつないで実行するだけ

# Run Test in the Cluster

Pod

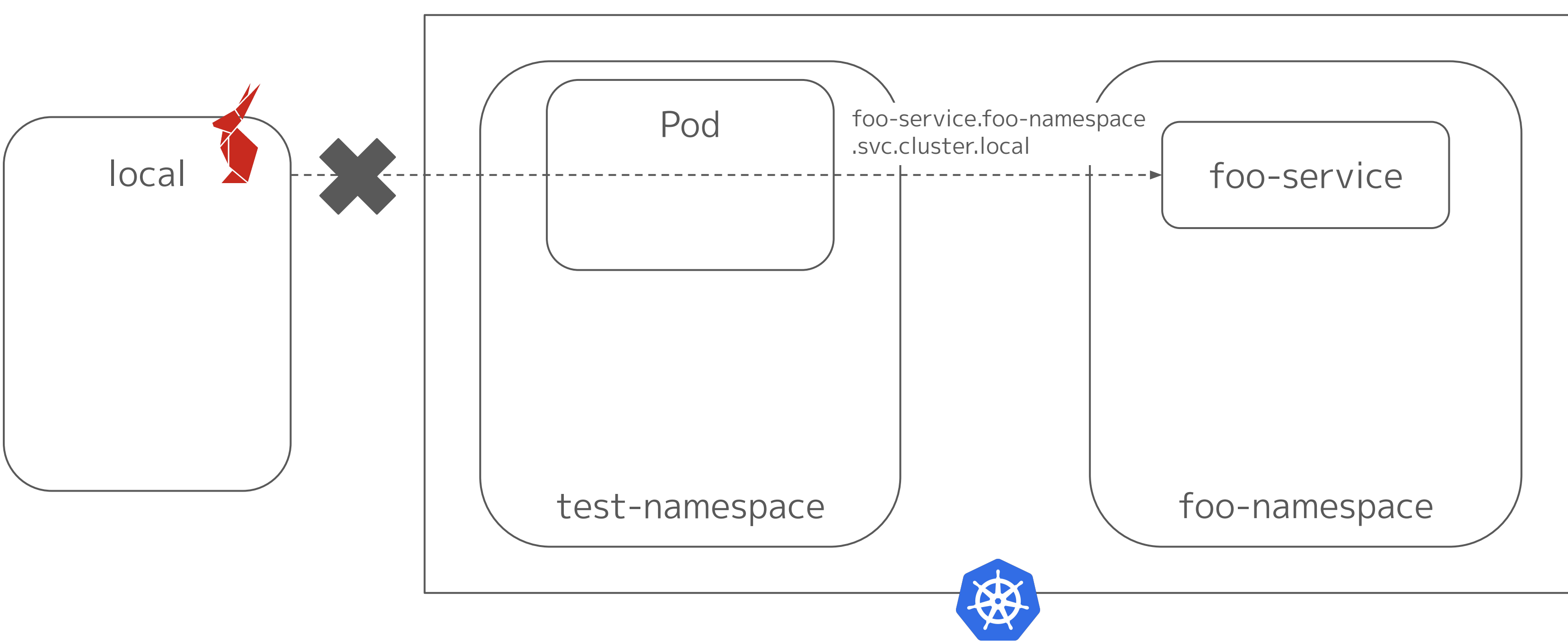
foo-service.foo-namespace
.svc.cluster.local

foo-service

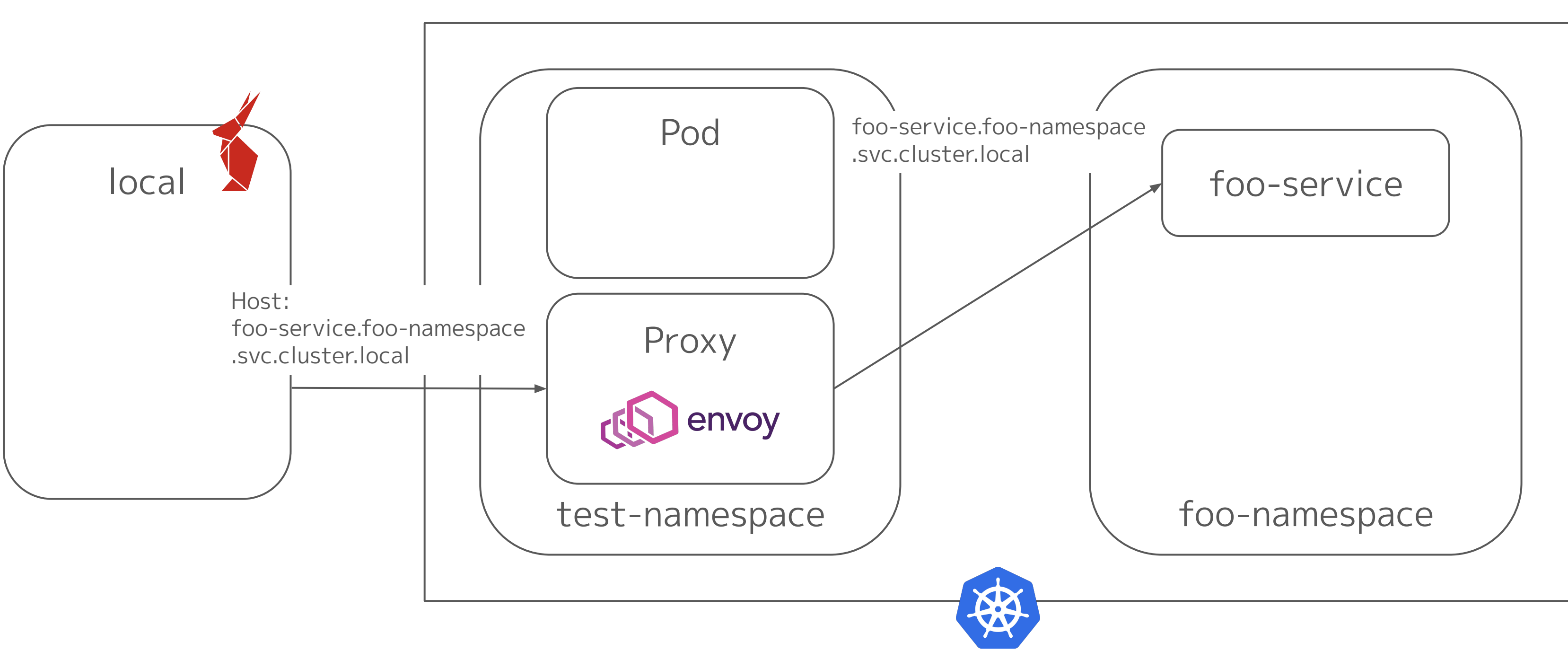
test-namespace

foo-namespace

# Run Test on Local



# Run Test on Local



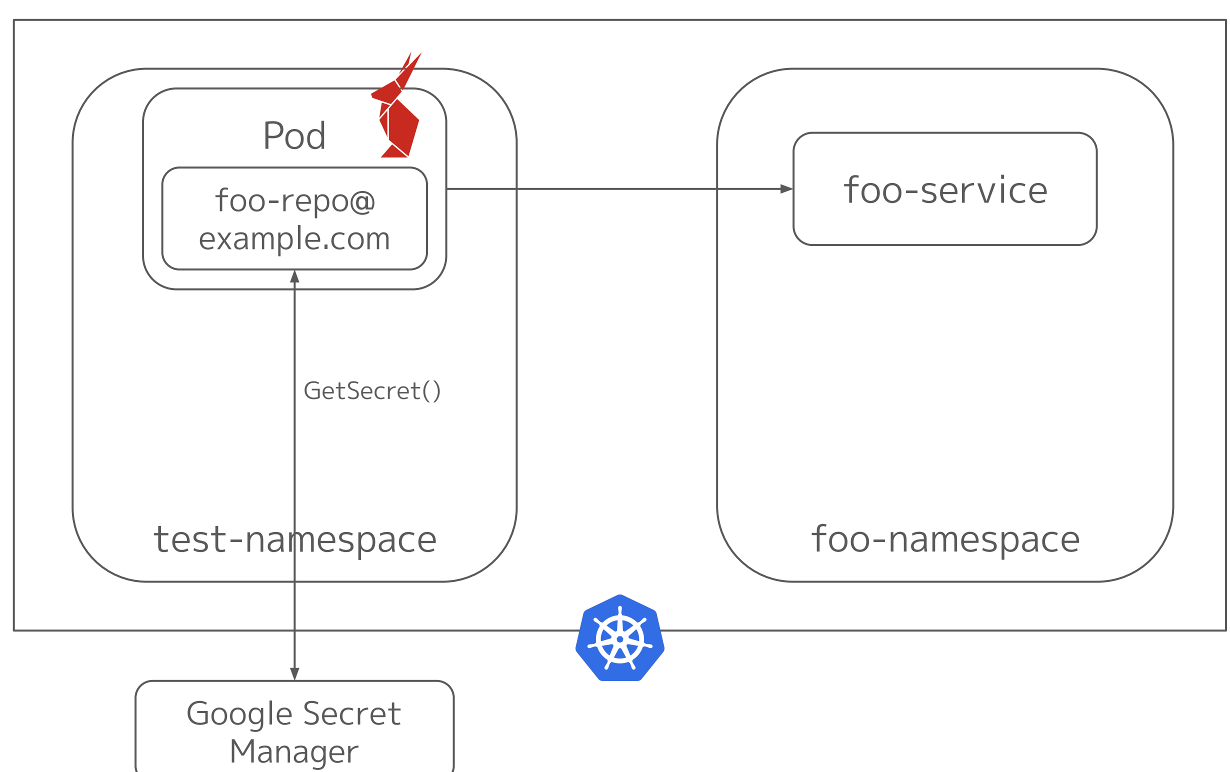
# Access Control

- レポジトリごとに専用のServiceAccountを作成、テストの実行時それを利用して権限管理をしている
  - たとえばSecretへのアクセス権など
- ローカルでも同じように動いてほしい
    - ローカル実行時は個人アカウント
      - 個人単位で別途権限を付与するのは煩雑
      - SA Keyを各PCにダウンロードして使うのはセキュリティ的に好ましくない
    - 個人アカウントからレポジトリのSAにimpersonateして実行
      - こちらも提供している関数を利用することで、SAの権限が必要な処理では自動でimpersonateするためローカルでも意識せず動く

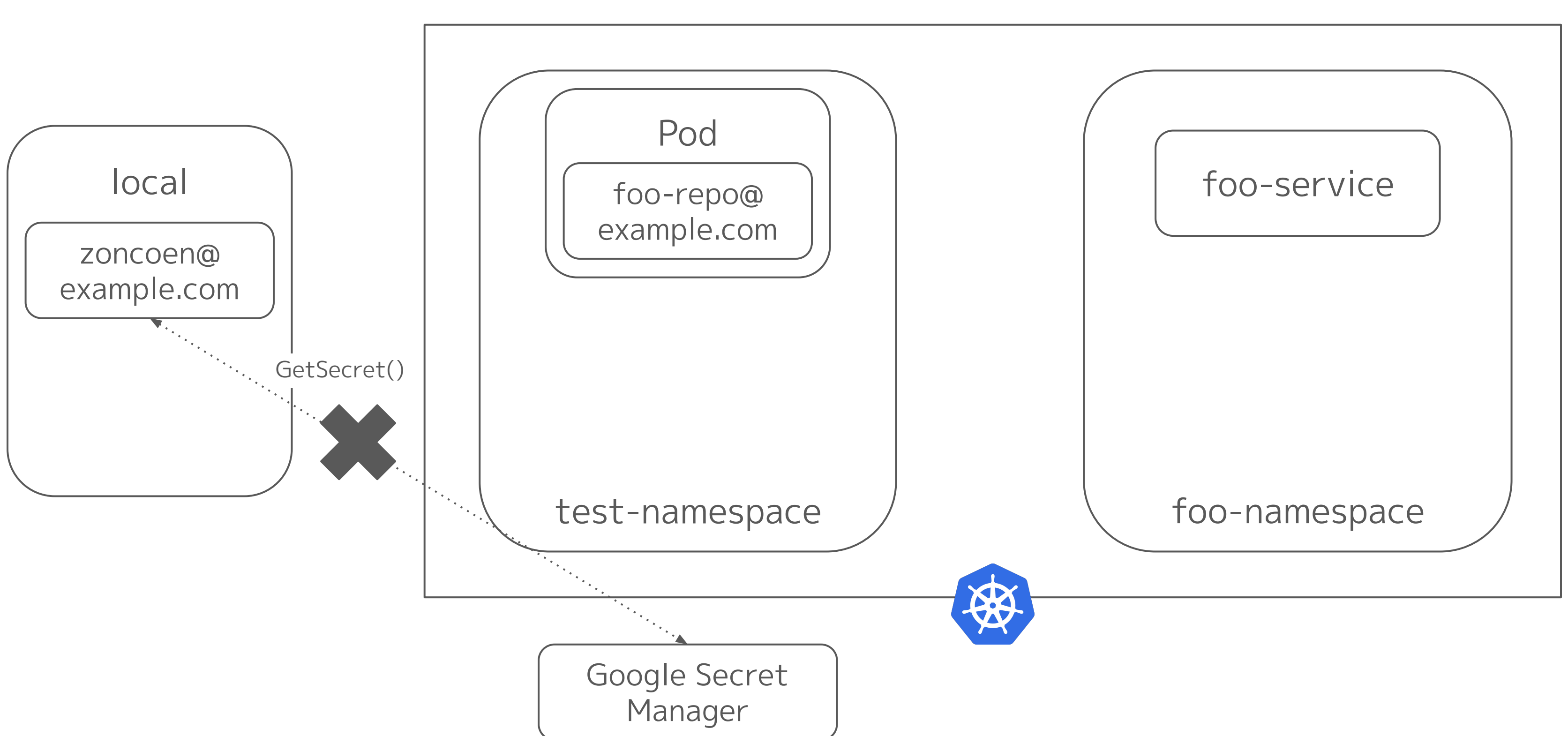
# SA Based Access Control

local

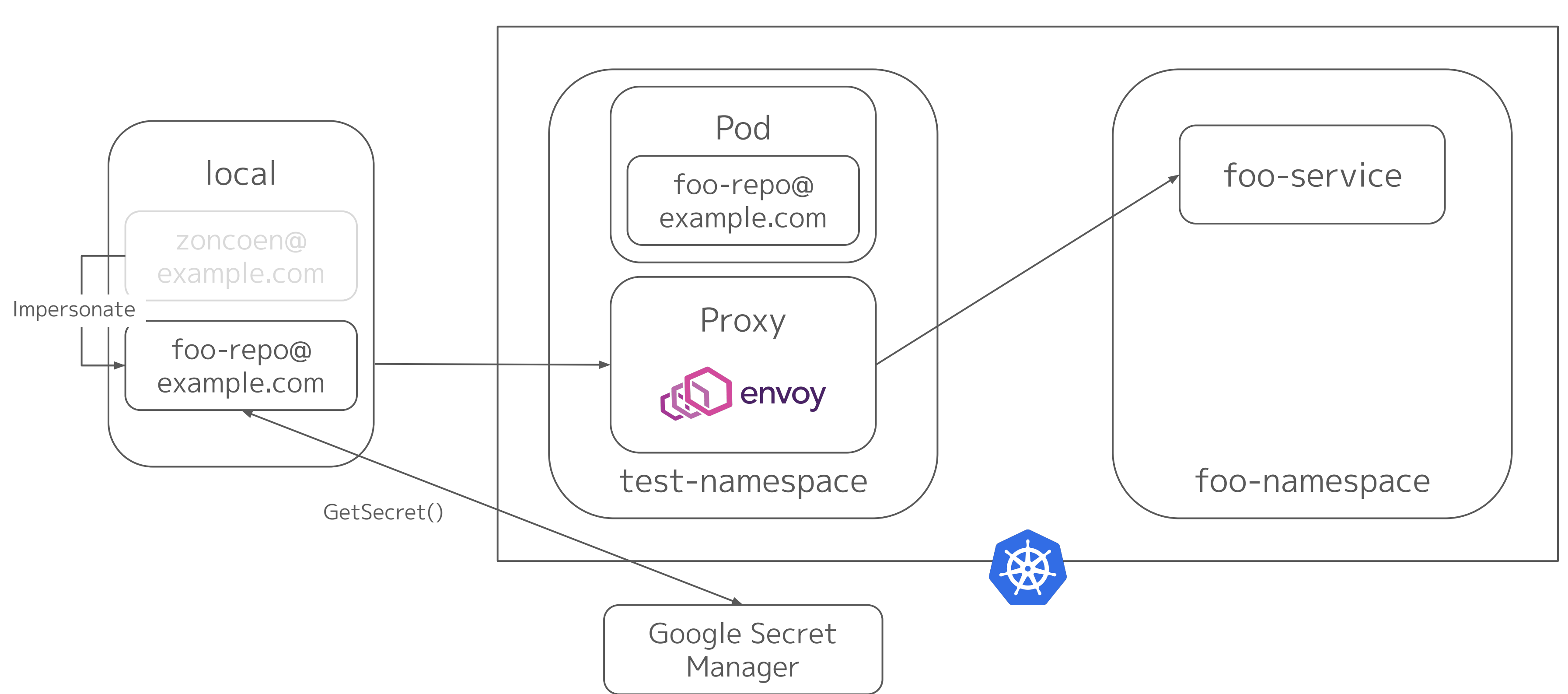
zoncoen@
example.com



# SA Based Access Control



# SA Based Access Control



# Conclusion

# Conclusion

- ScenarigoはGoで拡張できるのが大きな特徴
  - メルペイではその点を活かして社内向けのテストプラットフォームを開発し、QAなどで利用している
- 今後やりたいことアイデア
  - LSP対応（初期の頃からやりたいって言いながらできていない）
  - 他のプラグインメカニズムの追加
    - wasm対応してGo以外の言語でもプラグインを書けるようにしたい
  - 負荷試験が同じ形式のテストシナリオでできるように
  - テストシナリオ自動生成
  - 機能要望やFBがもしあれば @zoncoen にいただけるととても嬉しいです
    - 使わないとわからないことはたくさんある…難しい…

# Appendix - How to write test scenario

# Send HTTP requests

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
```

- テストシナリオは複数のステップをもつ
  - 1step = 1request

# Send HTTP requests

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: 1
```

- URL のクエリパラメータは直接書くか query フィールドが使える

# Send HTTP requests

```yaml
title: check /message
steps:
- title: POST /message
  protocol: http
  request:
    method: POST
    url: http://example.com/message
    body:
      message: hello
```

- POST リクエストでデータを送るには body フィールドを使う
  - デフォルトでは JSON にエンコードされる

# Send HTTP requests

- 他の形式でリクエストボディを送りたい場合は Content-Type ヘッダーを変更する

```yaml
title: check /message
steps:
- title: POST /message
  protocol: http
  request:
    method: POST
    url: http://example.com/message
    header:
      Content-Type: application/x-www-form-urlencoded
    body:
      message: hello
```

# Send HTTP requests

- 今リクエストボディのエンコードに対応している Content-Type
  - application/json
  - text/plain
  - application/x-www-form-urlencoded

# Check HTTP responses

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: 1
  expect:
    code: OK
    body:
      id: 1
      message: hello
```

- expect フィールドを使うことでレスポンスをテストすることができる
  - 実際のレスポンスと差異があればシナリオを中断してエラーを出力する

# Check HTTP responses

- 今レスポンスボディのデコードに対応している Content-Type
  - application/json
  - text/plain

# Retry

- 非同期処理を待つ場合などには step のリトライが利用できる

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
  retry:
    constant:
      interval: 1s
      maxElapsedTime: 5s
      maxRetries: 5
```

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
  retry:
    exponential:
      initialInterval: 500ms
      factor: 2
      jitterFactor: 0.5
      maxInterval: 2m
      maxElapsedTime: 5m
      maxRetries: 10
```

# Using variables

```yaml
title: check /message
vars:
  id: 1
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: '{{vars.id}}'
```

- 実行時に評価されるテンプレート文字列が使える
  - vars フィールドで定義した変数を参照して再利用できる

# Using variables

```yaml
title: check /message
steps:
- title: GET /message
  vars:
  - 1
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: '{{vars[0]}}'
```

- step scope な変数も定義できる
  - 他の step からは参照できない

# Reuse response value

- bind フィールドでレスポンスを保存して以降の step で利用できる

```yaml
- title: POST /message
  protocol: http
  request:
    method: POST
    url: http://example.com/message
    body:
      message: hello
  expect:
    code: OK
    body:
      message: '{{request.message}}'
  bind:
    vars:
      id: '{{response.id}}' < set
```

```yaml
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: '{{vars.id}}' < use
  expect:
    code: OK
```

# Environment variables

```yaml
title: check /message
steps:
- title: GET /message
  protocol: http
  request:
    method: GET
    url: '{{env.TEST_ADDR}}/message'
```

- 環境変数も参照できる

# Reuse scenario

- include で他のシナリオを step として再利用できる

```yaml
title: get message
steps:
- title: GET /message
  vars:
    message: hello
  include: create-message.yaml < include
  bind:
    vars:
      id: '{{vars.id}}' < set
- title: GET /message
  protocol: http
  request:
    method: GET
    url: http://example.com/message
    query:
      id: '{{vars.id}}' < use
```

```yaml
title: create message
vars:
  message: '{{vars.message}}' < input
steps:
- title: POST /message
  protocol: http
  request:
    method: POST
    url: http://example.com/message
    body:
      message: '{{vars.message}}'
  bind:
    vars:
      id: '{{response.id}}'  < output
```

# Plugin

```
$ scenarigo plugin build
```

```yaml
schemaVersion: config/v1
scenarios:
- scenarios
pluginDirectory: ./gen
plugins:
  date.so:
    src: ./plugins/date
```

```go
package main

import "time"

func Today() string {
  return time.Now().Format(Layout)
}
```

```yaml
title: echo
plugins:
  date: date.so
steps:
- title: POST /echo
  protocol: http
  request:
    method: POST
    url: '{{env.ECHO_ADDR}}/echo'
    body:
      message:
'{{plugins.date.Today()}}'
  expect:
    code: 200
```

# Send gRPC requests

```yaml
title: check Ping method
plugins:
  grpc: 'grpc.so'
steps:
- title: call Ping
  protocol: grpc
  request:
    client: '{{plugins.grpc.EchoClient}}'
    method: Ping
    body:
      message: hello
  expect:
    code: OK
    body:
      message: '{{request.message}}'
```

- proto から生成した Go のクライアントをプラグイン経由で使う
  - それ以外の基本的な使い方は HTTP の場合と同じ
  - server reflection service を使ってプラグイン使わないこともできるけどとりあえず未実装