DNSの可視化ダッシュボードを「コード」で作る
～長期運用を見据えたDashboard as Codeへの挑戦～

NTT東日本株式会社
先端テクノロジー部　ネットワーク技術部門
伊藤 雅子、根岸　亮太

目次

1.自己紹介
2.NTT東日本とフレッツサービス概要
3.自社の運用/開発体制
4.DNS監視と可視化へのモチベーション
5.DNS可視化開発の概要
6.GrafanaにおけるGUI利用課題
7.コード利用によるメリット
8.デモ紹介
9.まとめ

発表する人 自己紹介（根岸・伊藤）



●所属：
NTT東日本　先端テクノロジー部
ネットワーク技術部門　ネットワーク制御技術担当

●お仕事：
DNSサーバ、RADIUSサーバ、NTPの開発/検証/導入

●名前：
根岸　亮太

●住まい：
埼玉

●プライベート(趣味など)　：
ダーツ



●所属：
NTT東日本　先端テクノロジー部
ネットワーク技術部門　ネットワーク制御技術担当

●お仕事：
DNSサーバ、RADIUSサーバの開発/検証/導入

●名前：
伊藤　雅子

●住まい：
埼玉

●プライベート(趣味など)　：
子供（2人）と虫の採取、仮〇ライダーとウル〇ラマンの推し活

自社の運用/開発の体制

機能・品質を担保したプロダクトを作る「開発部門」と商用サービスの安定運用を担う「運用部門」が連携してシステムを運用

開発部門｜機能・品質の担保

- 要求整理、方式設計、検証、商用導入支援
- システム、成果物の提供
- 高度解析、原因・影響範囲の見極め
- 不具合修正、機能改善、更改対応
- 運用部門からのフィードバック反映
- 検証環境の維持管理

プロダクトの開発と維持管理

↑我々はこっちの人

連携ポイント

①成果物の提供
技術支援

②トラブル事象共有や
運用上のフィードバック

運用部門｜日々のサービス運営

- 商用装置の監視・故障検知、一次切り分け
- 利用影響の把握と関係部門への連絡
- 対処しきれない事象に関して情報を開発へ連携
- 現場視点の改善要望・運用課題をフィードバック

日々の監視や一次切り分け

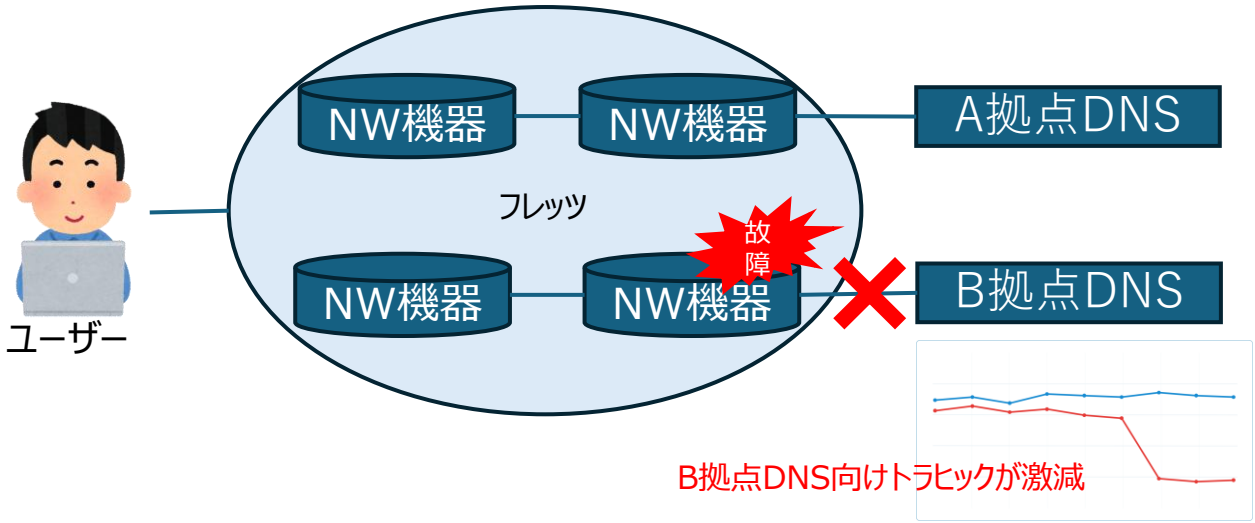
商用システムの運用は運用部門が担うが、提供したシステム・成果物の責任は開発部門が担う

⇒開発部門は最後の砦であり、提供したシステムの利用が終了するまで面倒をみなくてはならない

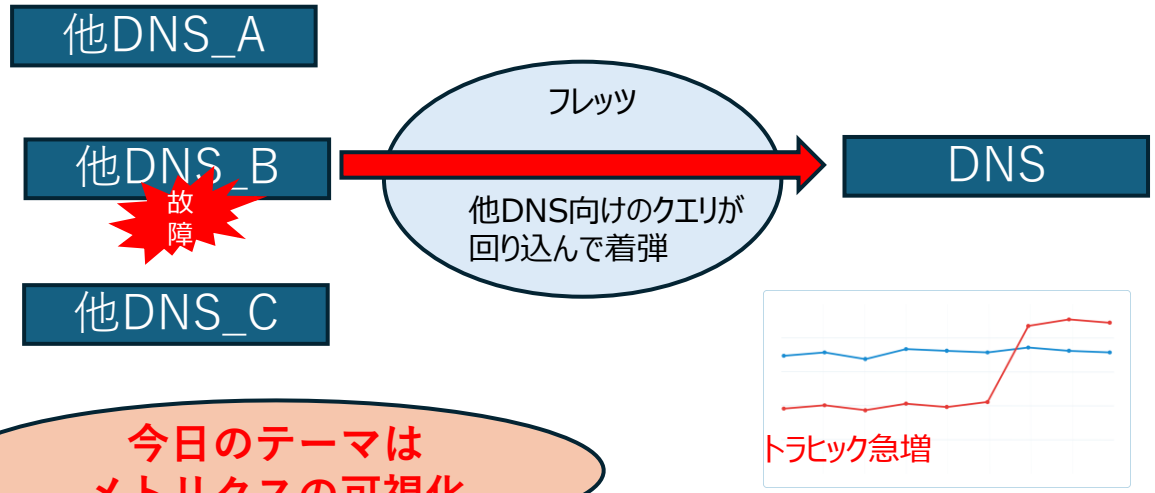
DNSの監視事情

ネットワーク障害時や他DNS故障時に運用部から要請を受けて開発部門が解析等を対応

ネットワーク障害時の例



他DNS故障時の例



今日のテーマは
メトリクスの可視化

■確認の流れ

1 **故障申告・調査依頼**
運用部から故障情報連絡、調査依頼

2 **区間別クエリ量**
拠点毎のDNSのトラヒック量を収集

3 **DNS健全性**
リソース、名前解決正常性の確認

4 **調査結果・回答**
DNSの状態および推測される原因について回答

運用部からの解析要請都度、各拠点のDNSサーバへログインし統計情報等を取得して加工

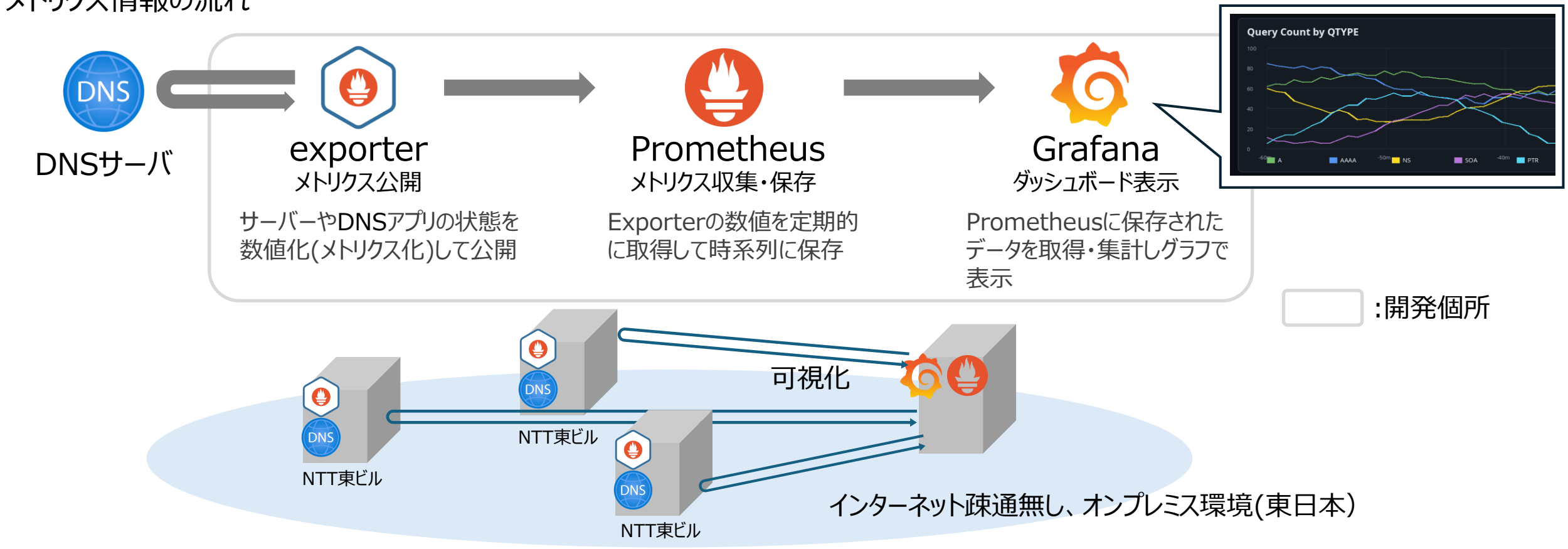
⇒**稼働、解析時間の増加**

DNSの状態やクエリ傾向だけでもすぐに確認できるよう「可視化開発」に着手

DNSの可視化開発の概要

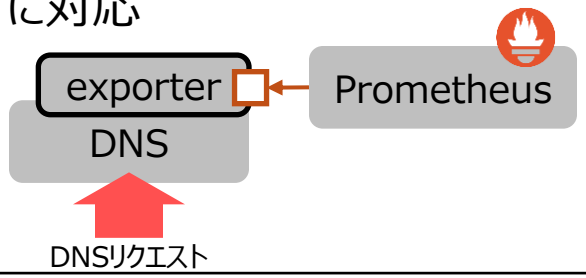
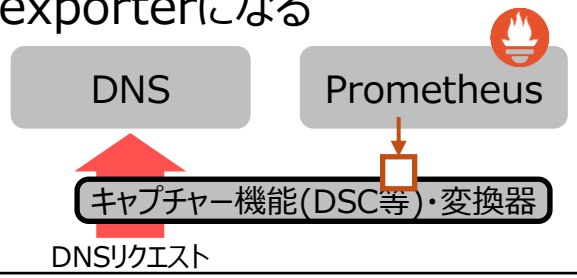
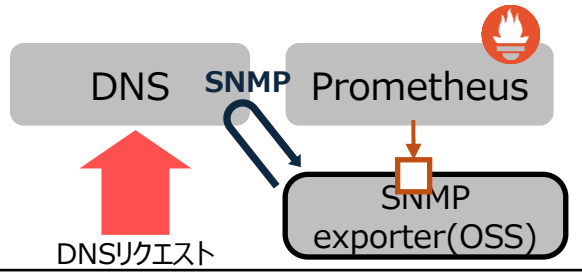
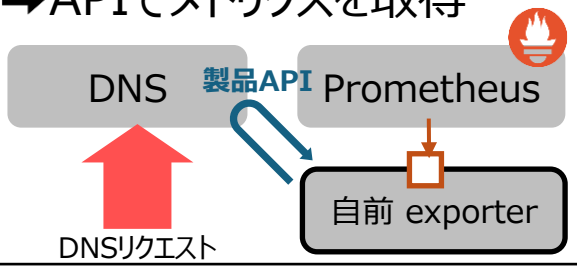
- 取得すべきメトリクスを事前に定義でき、スモールスタートで傾向把握から始められることから、Prometheus + Grafanaを採用
- DNSのクエリ情報およびCPU使用率などの統計情報をダッシュボード上で可視化する機能部(オブザーバビリティ機能部)を新規に開発中

メトリクス情報の流れ



参考.可視化構成の方式検討

- PrometheusでDNSのメトリクスを取得する方式は以下の4方式がある
- どの方式を選択するのかはDNS製品がどの方式に対応しているのかがキーポイント
- 今回私たちが開発しているDNS製品は一部自作が必要な製品であったため④を選択

方式	①	②	③	④
	DNSサーバ自体がexporterに対応 	paキャプチャーマシンがexporterになる 	SNMPでメトリクスを取得 	今回構成 exporterが存在しない ➡APIでメトリクスを取得 
メリ	直接メトリクスを取得可能 **➡構成がシンプル**	DNSサーバがexporterに**非対応でも使用可**	該当製品：市場にそれなりに存在している	必要な情報を**ピンポイントで取得**が可能
デメ	該当製品：市場には**少ない**	高性能なマシン(CPU,メモリ等)など**要求リソースが高い**	**取得情報が限られ**、設定が複雑な場合もある模様	自作が必要**➡開発・保守の手間**がかかる可能性
例	bindやunboundなどのOSS（オープンソースソフトウェア）	DSC、DNS-collector等	ネットワーク機器等	①～③以外
備考		NW上のDNSリクエスト等をキャプチャ・解析し、exporterとしてメトリクスを出力	SNMPによりDNSサーバーの情報を取得後、SNMP-exporterでメトリクス出力	自作のexporterやスクリプトでAPIからデータを取得し、exporterとしてメトリクス出力

可視化用のダッシュボード開発(Grafana開発）をGUIで実施するときの課題

背景： ゼロから**約50画面の開発**を実施※1。また短期間でフィードバックを反映しながら、段階的に作成・改善をする必要がある。
課題①： 表示期間やパネルの体裁を全画面で変えるには**一つずつ手作業になり、変更漏れや設定のばらつきが起きやすい**
課題②：「どこを、なぜ変えたか」を**差分でレビューできず、変更履歴も追いにくい**

※1.サーバ台数×サーバ種別×表メトリックで算出

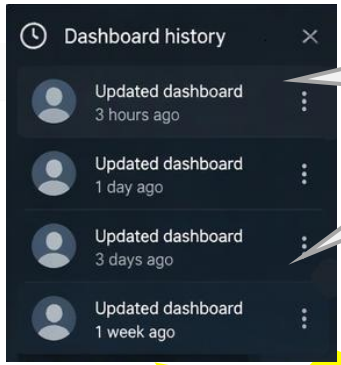
【課題①】開発者の課題※2



【課題②】レビュワーの課題※2







変更理由が不明

変更内容と
影響が不明

**変更意図や差分が分からず、
レビューや調査に時間を要す**

※2.生成AIにて一部画像作成

【解決策】Grafana Foundation SDKの導入

Grafana Foundation SDKを使えば、ダッシュボードを「コード」で定義し、JSONを自動生成可能に！

Grafana Foundation SDKとは？
・Grafana公式のOSSライブラリ（Go／TypeScript／Python／PHP／Javaに対応）を用いて、コードを記載して画面を作成するしくみのこと
・ダッシュボードをコードとして書くことで、Gitのような標準的なバージョン管理システムを使えば、変更をシームレスに追跡可能になる

画面クリックで設定

画面開発の流れ

【GUIで作成する場合】	【Foundation SDKで作成する場合】
①Grafana画面でパネルを追加し、クエリ・表示設定を手で入力	**① コードでダッシュボードを定義（ダッシュボード／パネル／クエリ）**
②保存後、JSONを手動でエクスポート	**② GitLabで差分レビュー・承認（マージリクエスト等）**
③JSONをGitLabに登録（差分が長く、読みにくい）	③作成したコードをコンパイルしダッシュボードJSONを自動生成
④AnsibleでJSONを配置	④ AnsibleでJSONを配置
⑤Grafanaが検知して自動で読込み	⑤ Grafanaが検知して自動で読込み
▲直すときは画面ごとに①～③を繰り返す	◎ 共通部分は1か所直せば全画面に反映

型つきコードで設定
➡GitLab・IDEと組み合わせてレビュー漏れ防止！

【解決策】Grafana Foundation SDK導入の具体的な効果
～Dashboard as Codeで得られるもの～

コードで定義すれば、差分もGitLab上でレビューでき、かつ全画面への変更は1回の修正で反映可能

修正前

```
// ===== ⚙ 全画面共通の設定（ここを直せば、全画面に反映）=====
const (
        timeFrom = "now-24h" // 表示期間：直近24時間
        refresh  = "1m"      // 自動更新：1分ごと
        servfailLimit = 10   // SERVFAILのしきい値（これを超えたら赤）
)

// ===== ⚙ パネルを「部品」として定義 =====
// 部品A：クエリ数の推移（折れ線グラフ）
func queryRatePanel(server string) *timeseries.PanelBuilder {
        return timeseries.NewPanelBuilder().
                Title("クエリ数（秒あたり）").
                Unit("reqps").
                Span(18).Height(8).
                WithTarget(
                        prometheus.NewDataqueryBuilder().
                                Expr(fmt.Sprintf(
                                        `sum by (type)
  (rate(bind_incoming_queries_total{server="%s"}[5m]))`, server)).
                                LegendFormat("{{type}}"), // A, AAAA, MX…ごとに線を分け
  る
```

コード修正

修正後

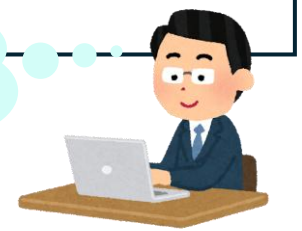
```
// ===== ⚙ 全画面共通の設定（ここを直せば、全画面に反映）=====
const (
        timeFrom = "now-7d"  // 表示期間：直近7日間
        refresh  = "1m"      // 自動更新：1分ごと
        servfailLimit = 10   // SERVFAILのしきい値（これを超えたら赤）
)

// ===== ⚙ パネルを「部品」として定義 =====
// 部品A：クエリ数の推移（折れ線グラフ）
func queryRatePanel(server string) *timeseries.PanelBuilder {
        return timeseries.NewPanelBuilder().
                Title("クエリ数（秒あたり）").
                Unit("reqps").
                Span(18).Height(8).
                WithTarget(
                        prometheus.NewDataqueryBuilder().
                                Expr(fmt.Sprintf(
                                        `sum by (type)
  (rate(bind_incoming_queries_total{server="%s"}[5m]))`, server)).
                                LegendFormat("{{type}}"), // A, AAAA, MX…ごとに線を分け
  る
```

マージリクエスト

GitLab

```
-  timeFrom = "now-24h" // 表示期間：直近24時間
+  timeFrom = "now-7d"  // 表示期間：直近7日間
```

コメントした箇所が確かに修正されているな



単純な件数表示だけでなく、警告と異常を色分けしたい。10件以上は黄色、30件以上は赤色にし、説明文も全画面に追加してほしい。

要望部



共通部品の修正例（Go言語）　servfailPanel() を1か所修正するだけ

修正前

```
func servfailPanel(server string)
  *stat.PanelBuilder {
  return stat.NewPanelBuilder().
    Title("SERVFAIL（直近5分）").
    Span(6).Height(8).
    Thresholds(
      dashboard.NewThresholdsConfigBuilder().
       Mode(dashboard.ThresholdsModeAbsolute).
       Steps([]dashboard.Threshold{
         {Value: nil, Color: "green"},
         {Value: cog.ToPtr[float64](10),
            Color: "red"},
       }),
    ).
    WithTarget(
      prometheus.NewDataqueryBuilder().
       Expr(fmt.Sprintf(
         `sum(increase(bind_responses_total{
          server="%s",result="SERVFAIL"}[5m]))`,
          server)),
  }
```

変更点

- 説明文を追加
- しきい値を変更
  10件以上：黄色
  30件以上：赤

修正後

```
func servfailPanel(server string)
  *stat.PanelBuilder {
  return stat.NewPanelBuilder().
    Title("SERVFAIL（直近5分）").
    Description("直近5分のSERVFAIL応答数、
    10件以上は警告、30件以上は異常").
    Span(6).Height(8).
    Thresholds(
      dashboard.NewThresholdsConfigBuilder().
       Mode(dashboard.ThresholdsModeAbsolute).
       Steps([]dashboard.Threshold{
        {Value: nil, Color: "green"},
        {Value: cog.ToPtr[float64](10), Color: "yellow"},
        {Value: cog.ToPtr[float64](30), Color: "red"},
       }),
    ).
    WithTarget(
      prometheus.NewDataqueryBuilder().
       Expr(fmt.Sprintf(
         `sum(increase(bind_responses_total{
          server="%s",result="SERVFAIL"}[5m]))`,
          server)),
  }
```

パネルの説明文を追加（全画面に表示される）

しきい値を2段階に変更
10件以上：黄色
30件以上：赤

すべての画面がservfailPanel(server)を利用するため、**<u>共通部品を1か所修正してJSONを再生成すると複数の画面コードへ反映</u>**される

※.生成AIにて一部画像作成

デモの流れと考え方

① コードで定義
ダッシュボードを
プログラムコードで書く

② Gitでレビュー
コードレビュー

③ JSON化
Grafanaが読める
JSON形式に変換

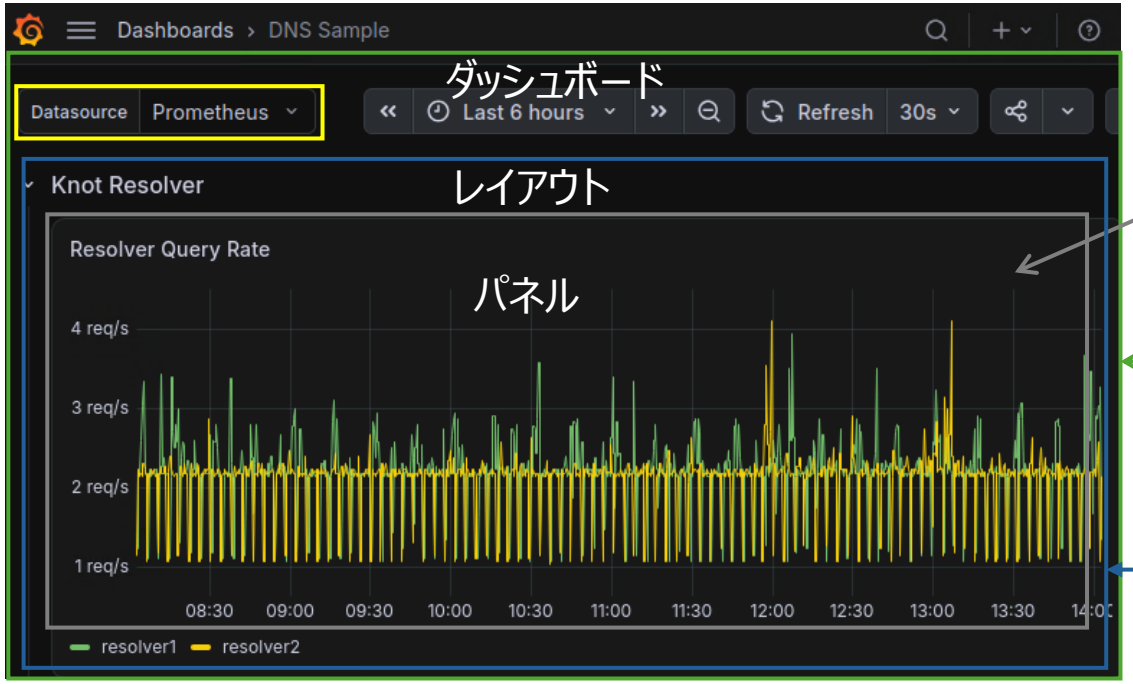
④Ansibleで配置
JSONをAnsilbe経由
でGrafanaに配置

⑤Grafanaへ登録
provisioner
で読み込み

押さえておきたい考え方

- 画面上の設定項目（タイトル、データソース、クエリなど）が、SDKの関数（～Builder にそのまま対応している）
- メソッドチェーン(“.”でメソッドをつなぐ)で積み上げ、最後に Build()で、JSONを出力する

ダッシュボード



goのコード

```
const (
    dnsSamplePanelResolverQPS    = 1
    resolverQPSElement           = "resolver-qps"
)

resolverQueryRate := dashboardv2.NewPanelBuilder().
    Id(dnsSamplePanelResolverQueryRate).
    Title("Resolver Query Rate").
    Visualization(timeseries.NewVisualizationV2Builder().
        Unit("reqps").
        Tooltip(defaultTooltip()).
        Legend(defaultLegend())).
    Data(dashboardv2.NewQueryGroupBuilder().
        Target(dashboardv2.NewTargetBuilder().Query(promQuery().
            Expr(`sum by (instance) (rate(resolver_request_total{` + resolver + `}[$__rate_interval]))`).
            LegendFormat("{{instance}}"))))
```

パネル定義

```
d, err := dashboardv2.NewDashboardBuilder("DNS Sample").
    Tags([]string{"dns", "sample"}).
    TimeSettings(dashboardv2.NewTimeSettingsBuilder().Timezone("browser").From("now-6h").To("now").AutoRefresh("30s")).
    CursorSync(dashboardv2.DashboardCursorSyncCrosshair).
    DatasourceVariable(dashboardv2.NewDatasourceVariableBuilder(datasourceVariable).
        Label("Datasource").
        PluginId("prometheus")).
    Panel(resolverQueryRateElement, resolverQueryRate).
    RowsLayout(dashboardv2.NewRowsBuilder().
        Row(dashboardv2.NewRowBuilder().
            Title("Knot Resolver").
            AutoGridLayout(dashboardv2.AutoGrid().
                MaxColumnCount(1).
                ColumnWidthMode(dashboardv2.AutoGridLayoutSpecColumnWidthModeStandard).
                RowHeightMode(dashboardv2.AutoGridLayoutSpecRowHeightModeStandard).
                WithItem(resolverQueryRateElement)))).
    Build()
```

ダッシュボード定義

データソース

レイアウト定義

実際のデモ画面①②

① コードで定義
ダッシュボードを
プログラムコードで書く

▶

② Gitでレビュー
コードレビュー

▶

③ JSON化
Grafanaが読める
JSON形式に変換

▶

④Ansibleで配置
JSONをAnsilbe経由
でGrafanaに配置

▶

⑤Grafanaへ登録
provisioner
で読み込み

コードでレビューする場合は、コードの差分を確認する
（変わっていない事の確認はGit任せでOK）



GUIでは、多様なプロパティを確認する
（変わっている点と、変わってない点の両方）



実際のデモ画面③

① コードで定義
ダッシュボードを
プログラムコードで書く

② Gitでレビュー
コードレビュー

③ JSON化
Grafanaが読める
JSON形式に変換

④Ansibleで配置
JSONをAnsilbe経由
でGrafanaに配置

⑤Grafanaへ登録
provisioner
で読み込み

```
go run cmd/generate(ソースコードの配置場所)
```

SDKコード(6パネルで約300行)

```
238         // レイアウトを定義
239         RowsLayout(dashboardv2.NewRowsBuilder().
240             Row(dashboardv2.NewRowBuilder().
241                 Title("Summary").
242                 AutoGridLayout(dashboardv2.AutoGrid().
243                     MaxColumnCount(4).
244                     ColumnWidthMode(dashboardv2.AutoGridLayoutSpecColumnWidthModeNarrow).
245                     RowHeightMode(dashboardv2.AutoGridLayoutSpecRowHeightModeShort).
246                     WithItem(resolverQPSElement).
247                     WithItem(resolverServfailRateElement).
248                     WithItem(pdnsAuthQPSElement).
249                     WithItem(pdnsAuthServfailRateElement))).
250             Row(dashboardv2.NewRowBuilder().
251                 Title("Knot Resolver").
252                 AutoGridLayout(dashboardv2.AutoGrid().
253                     MaxColumnCount(1).
254                     ColumnWidthMode(dashboardv2.AutoGridLayoutSpecColumnWidthModeStandard).
255                     RowHeightMode(dashboardv2.AutoGridLayoutSpecRowHeightModeStandard).
256                     WithItem(resolverQueryRateElement).
257                     WithItem(resolverResponseCodesElement))).
258             Row(dashboardv2.NewRowBuilder().
259                 Title("pdns-auth").
260                 AutoGridLayout(dashboardv2.AutoGrid().
261                     MaxColumnCount(1).
262                     ColumnWidthMode(dashboardv2.AutoGridLayoutSpecColumnWidthModeStandard).
263                     RowHeightMode(dashboardv2.AutoGridLayoutSpecRowHeightModeStandard).
264                     WithItem(pdnsAuthQueryRateElement).
265                     WithItem(pdnsAuthResponseCodesElement)))).
266         Build()
```

JSON:約1,000行

```
{
  "apiVersion": "dashboard.grafana.app/v2",
  "kind": "Dashboard",
  "metadata": {
    "name": "dns-sample"
  },
  "spec": {
    "annotations": [],
    "cursorSync": "Crosshair",
    "editable": true,
    "elements": {
      "pdns-auth-qps": {
        "kind": "Panel",
        "spec": {
          "id": 3,
          "title": "pdns-auth QPS",
          "description": "Authoritative queries per second across both transports.",
          "links": [],
          "data": {
            "kind": "QueryGroup",
            "spec": {
              "queries": [
                {
                  "kind": "PanelQuery",
                  "spec": {
                    "query": {
                      "kind": "DataQuery",
                      "group": "prometheus",
                      "version": "v0",
                      "datasource": {
                        "name": "$datasource"
                      },
                      "spec": {
                        "expr": "sum(rate(pdns_auth_udp_queries{job=\"scrapeConfig/monitoring/pdns-auth-external\"}[$__rate_interval
                        "instant": true,
                        "legendFormat": "QPS",
                        "range": false
```

実際のデモ画面④⑤

① コードで定義
ダッシュボードを
プログラムコードで書く

② Gitでレビュー
コードレビュー

③ JSON化
Grafanaが読める
JSON形式に変換

④Ansibleで配置
JSONをAnsilbe経由
でGrafanaに配置

⑤Grafanaへ登録
provisionerで
読み込み

Grafanaの自動プロビジョニング機能で、ダッシュボードのパスを任意のフォルダに指定し
生成したダッシュボードファイル(JSONファイル)を該当フォルダに配置することで、ダッシュボードとして表示する
→この手順をAnsible化することで、GUI操作なしでダッシュボードをデプロイ

自動プロビジョニング設定(例)

/etc/grafana/provisioning/dashboards/default.yaml

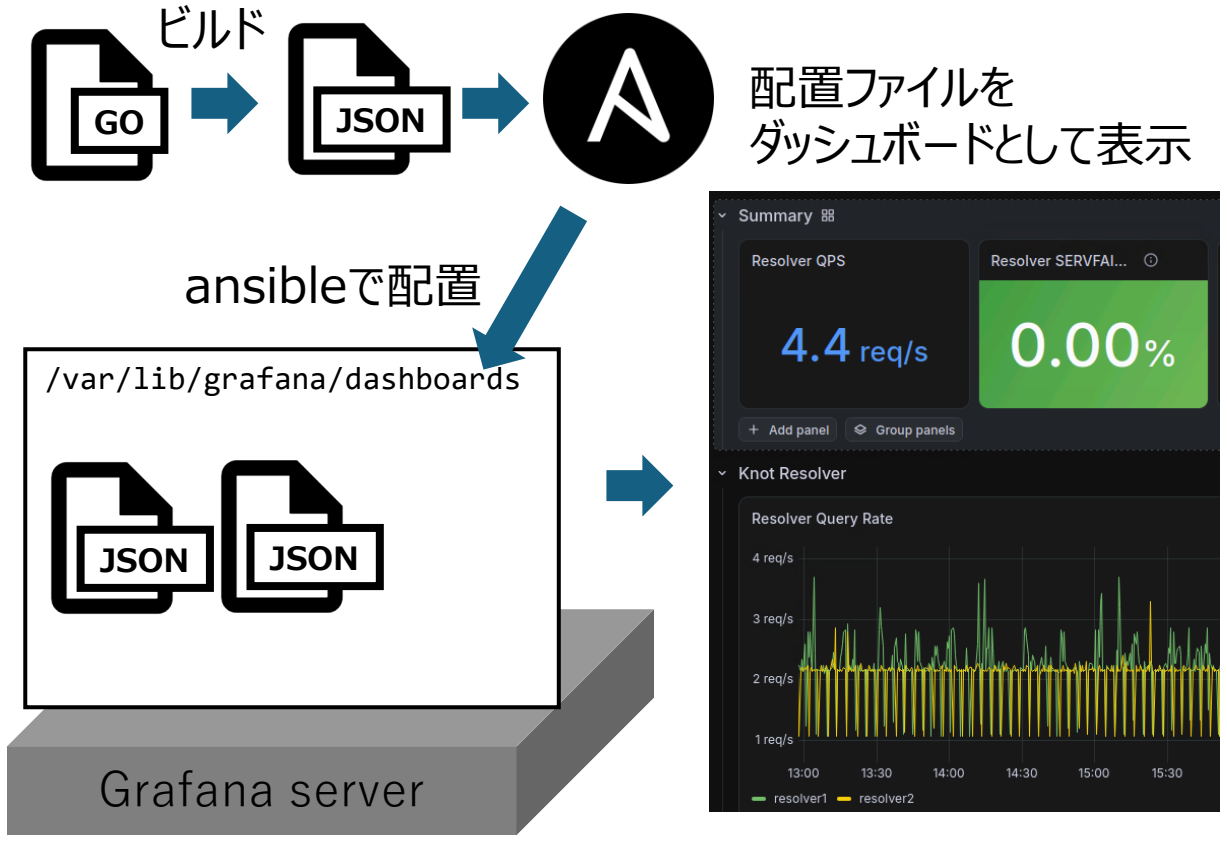
```yaml
apiVersion: 1
providers:
  - name: default
    type: file
    options:
      path: /var/lib/grafana/dashboards
      foldersFromFilesStructure: true
```

ダッシュボード定義

/etc/grafana/provisioning/datasources/prometheus.yaml

```yaml
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    url: https://prometheus.example.com
    isDefault: true
```

データソース




Grafana
Dashboards › DNS Sample
Search... ctrl+k
Sign in
Starred
Dashboards
Explore
Drilldown
Alerting
Connections
Add new connection
Data sources
Administration
General
Default preferences
Provisioning
Plugins and data
Plugins
Correlations
Users and access
Users
Teams
Service accounts
Datasource Prometheus
Last 6 hours
Refresh 30s
Edit
DNSクエリ量のスナップショット
Summary
Resolver QPS
4.4 req/s
Resolver SERVFAIL Rate
0.00%
pdns-auth QPS
4.7 req/s
pdns-auth SERVFAIL Rate
0.00%
Knot Resolver
Resolver Query Rate
DNSクエリ量の時系列データ
4 req/s
3 req/s
2 req/s
1 req/s
08:30 08:45 09:00 09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00 12:15 12:30 12:45 13:00 13:15 13:30 13:45 14:00 14:15
resolver1
resolver2
Resolver Response Codes
6 req/s
5 req/s
4 req/s
3 req/s

まとめ

- DNSの解析時間の削減やトラヒック傾向把握の簡易化をモチベーションに可視化開発に着手

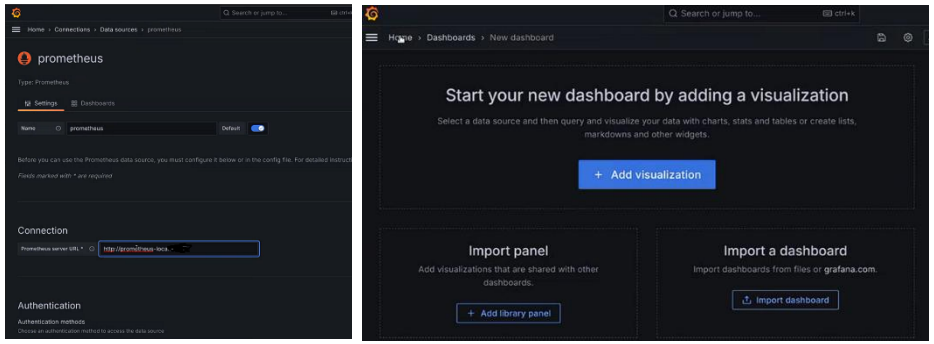
- 今回の開発スタイル的にCodeによるダッシュボード作成が有効と判断し挑戦
  -多数のダッシュボードの変更作業の負担軽減
  -レビュワーの負担軽減

- 可視化のやり方に関して知見があればぜひ伺いたいです！
  -どんなツールを利用していますか？
  -可視化の対象は何にしているか、それにしている理由はなにか
  -ダッシュボードの更新ってどれくらいの頻度でありますか？
  -ダッシュボード作成で意識していること、工夫していることは？

(参考)Grafana Foundation SDKとは？

- Grafanaダッシュボードの開発・修正・検証では、変更履歴の管理やレビューをしやすくするため、GUI操作による手作業ではなく、Grafana Foundation SDKを用いたコード管理方式を検討している。

【(A)GUI操作でGrafanaダッシュボードを設定構築・修正】



GUI操作でダッシュボードの設定

【(B)Grafana Foundation SDKを用いた設定構築・修正】

Go言語等でダッシュボードの設計内容をコードに書く → SDKでGrafana用ダッシュボード(のJSON)を生成 → Grafanaへ反映

	メリット	デメリット
(A)	・直感的に操作できる ・すぐに作成、修正できる ・画面を見ながら調整しやすい ・プログラミング知識がなくても扱いやすい	・手作業のため設定漏れや入力ミスが起きやすい ・同じ修正を複数箇所へ反映するのが大変 ・変更履歴やレビューがしづらい ・環境間で設定差分が出やすい ・ダッシュボード数が増えると保守が大変
(B)	・設定をコードとして管理できる ・Git等のバージョン管理ツールで変更履歴を追いやすい ・レビューしやすい ・同じ設定を複数環境へ展開しやすい ・共通部品化、テンプレート化しやすい ・CI/CDによる自動化がしやすい	・SDKやプログラミングの学習が必要 ・初期導入に手間がかかる ・小規模・単発用途では過剰になる場合がある ・画面上で細かく見た目を調整する作業はGUIの方が楽な場合がある