Процесс создания метрик я пытался стандартизировать, насколько это возможно, поэтому если что-то непонятно на каком-то этапе процессе создания, рекомендую смотреть на то, как это сделано в других метриках, отличаться почти не должно.

Всего есть два вида метрик.

**Основанные на снапшотах.**

Все метрики, которые выполняются запросы к API (например, к Github API), должны быть основаны на снапшотах. Запросы к API должны выполняться в “core”-сервере.

**Рассчитываемые динамически на фронтенде.**

Такие метрики не выполняют никаких запросов к API и вместо этого преобразовывают данные, полученные из своих зависимостей.

Для этих метрик в “core”-сервере ничего добавлять не нужно.

Секция “core”:

Здесь метрики нужно добавлять только если нужны API-запросы к гитхабу или другим сервисам.

1. Создать функцию в папке “metric/core/tasks”:

func CommitsMetric(task models.Task, repo \*repositories.SnapshotRepository) {

Она должна принимать два аргумента:

* task: models.Task - объект, у которого нам интересно прежде всего свойство Data - массив параметров метрики и “ресурса”, которые передаются в том же формате, что и в конфигах в “server”. Например, API-ключи, API-эндпоинт
* repo: \*repositories.SnapshotRepository - при помощи этого репозитория сохраняются результаты сбора данных.

Эта функция будет вызываться через установленный промежуток времени (см. UpdateRate в секции “server”).

По окончанию работы нужно записать результаты в базу данных при помощи аргумента “repo” (доступные методы можно посмотреть в “metric/core/repositories/snapshotRepository.go” и при необходимости дописать новые).

Для записи передаются объекты models.Snapshot, в которых содержатся:

* Metric - название метрики
* Data - данные, которые хотим сохранить. Каждая метрика сама определяет формат. В секции “admin” описывается, как распознавать этот формат на фронтенде с помощью zod. Сохраняется как сериализованная JSON-строка
* Groups - теги. Используются для того, чтобы получать снапшоты через gRPC. Определяются внешним сервером (секция “server”), здесь их просто нужно передать в исходном виде из аргумента task.groups.  
  На данный момент используются только теги формата “resource:Имя ресурса” и “project:Имя проекта”. При необходимости сгруппировать и получать одним gRPC-запросом снапшоты по какому-то другому принципу можно добавлять любые другие теги.
* Params - объект со свойствами (“name” и “value”). Используются для создания более эффективных запросов к уже сохранённым снапшотам в БД.  
  Например, таск “Commits” сохраняет все найденные изменения в файлах и id и commit\_sha всех файлов сохраняет в Params. При необходимости по этому id можно получить все изменения конкретного файла, которые у нас сохранены в БД, а по commit\_sha - все файлы, изменённые в коммите.
* Error - пустая строка, если всё хорошо. Если были проблемы, записать текст ошибки, чтобы во фронтенде отображалось, что в какой-то момент что-то пошло не так.
* IsPublic - берётся напрямую из task. Определяет, передавать ли данные по этому снапшоту через gRPC. Например, для CommitFiles указывается false, потому что мы не хотим перегружать клиент перекидыванием истории всех файлов репозитория, а вместо этого пользоваться CommitFiles только в пределах “core”-сервера

Для более удобного перевода данных в models.Snapshot в “metric/core/task/helpers.go” определена функция:

func taskToSnapshot(task models.Task, data string, err string, params []models.SnapshotParam) \*models.Snapshot

, где models.SnapshotParam: { name: string, value: string }

1. В файле metric/core/tasks/list.go добавить созданную функцию в список. Например:
2. "Commits": CommitsMetric,

Ключ - название метрики, которое определяется в секции “server”

Значение - функция из шага 1

Секция “server”:

1. В файле “metric/server/src/metric/config/instances/metricNames.ts” добавить в enum MetricName название метрики. Например,

TotalCommits = 'Total Commits',

, где значение в кавычках - имя, которое будет использоваться для отображения в “admin” и в шаге 2 в “core”

1. В файле “metric/server/src/metric/config/instances/metricConfig.ts” добавить настройки метрики:

Для этого в объект metricConfig нужно добавить новое свойство, ключ которого - название метрики, определённое на шаге 1.

Значение - объект со свойствами:

* dependencies - string[] - массив названий метрик, от которых зависит данная. Зависимость означает, что этот список метрик будет автоматически создаваться при создании данной метрики, а данные по этим зависимостям будут напрямую передаваться данной метрике во фронтенде (см. секцию “admin”)
* snapshotBased - boolean - указывает на то, использует ли метрика снапшоты.  
  True - метрика вычисляется в “core”-сервере  
  False - не нужно делать обращений к “core”-серверу, все вычисления будут происходить на фронтенде при помощи данных из dependencies
* platform - string - название платформы, к которой относится данная метрика (например, GitHub). Названия берутся из enum PlatformName. При необходимости этот enum можно пополнять другими платформами.
* params - массив параметров метрики. Каждый параметр - объект со свойствами:
  + type - тип из enum MetricParamType. Типы определены в “metric/server/src/metric/config/types.ts”.
  + name - string - название параметра.
  + Остальные свойства зависят от выбранного type

Отдельно стоит отметить, что есть зарезервированные названия параметров:

* “isGraded” - type: MetricParamType.boolean - нужно ли оценивать эту метрику.
* “gradeWeight” - type: MetricParamType.text - вес оценки в контексте определения итоговой оценки проекта. 1 - вес по умолчанию. 2 - метрика в два раза больше влияет на итоговую оценку. 0.5 - в два раза меньше, и т.д.
* “updateRate” - type: MetricParamType.duration - то, насколько часто нужно обновлять метрику в “core”-сервере. Обязательный параметр для метрик, основанных на снапшотах
* “weight” - type: MetricParamType.text - вычислительный вес метрики. В “core”-сервере указывается максимальный суммарный вес одновременно вычисляемых метрик.  
  Создан на случай, если появится метрика, которая, например, очень сильно будет забивать оперативку. В этом случае можно поставить weight равный максимальному суммарному весу в “core”, чтобы остальные задачи окончания этой задачи, а не выполнялись параллельно.

Секция “admin”

1. Добавить имя метрики в конфиг

В файле “metric/admin/src/entities/Metric/config/instances/types.ts” добавить метрику в enum MetricName

В формате: ИмяМетрики = “Имя Метрики”

1. Создать папку метрики

Внутри папки “metric/admin/src/entities/Metric/instances”.

Название папки в формате “ИмяМетрики”

Всё, что внутри этой папки будет происходить, необязательно делать ровно как написано в инструкции, но для всех остальных метрик был следующий формат:

* “/index.ts” импортирует и сразу же экспортирует все файлы, которые должны быть доступны для импорта глобально
* “/types” - typescript-интерфейсы и типы
* “/hooks” - React-хуки
* “/model” - Бизнес-логика. Например, функция преобразования данных в нужный формат, функция для высчитывания оценки
* “/ui” - Визуальное представление метрики
* “/utils” - Вспомогательные функции

1. Определить интерфейсы данных метрики

3.1. (Если метрика использует снапшоты)  
Если метрика основана на снапшотах, то необходимо определить “Schema” с использованием библиотеки “zod”. Например, как это сделано в метрике “Commits” или “Issues”.Далее нужно будет преобразовать эту схему в тип TypeScript. Например (для метрики “Commits”):

export type Commits = z.infer<typeof CommitsSchema>;

3.2.Определить и экспортировать интерфейс метрики, например (для метрики Commits):

export interface CommitsMetric extends IGenericMetric {

name: MetricName.Commits;

data: Commits;

}

Name взять из MetricName, который определили на шаге 1.

Data взять из шага 3.1. Если метрика не использует снапшоты, просто не указывать свойство “data”.

1. Включить интерфейс из шага 3.2 в список метрик

В файле “metric/admin/src/entities/Metric/config/instances/types.ts”

Дополнить тип IMetric созданным на шаге 3.2 интерфейсом

1. Добавить метрику в convertMetric

В файле “metric/admin/src/entities/Metric/config/instances/convertMetric.ts”

Этот файл предназначен для валидации полученных с сервера данных на основе схем библиотеки Zod из шага 3.2.

При этом даже если метрика не использует данные снапшотов, добавить её всё равно сюда нужно. Для этого в switch-case нужно добавить, например:

case MetricName.Commits:

return {

...metric,

name: MetricName.Commits,

data: convertData(CommitsSchema, metric.data),

};

Name взять из шага 1.

В data вместо CommitsSchema взять схему, определённую на шаге 3.1. Если метрика не основана на снапшотах, просто не указывать data.

1. Добавить пользователей в список вариантов для выбора в фильтрах  
   Пользователи для фильтров определяются динамически на основе данных из метрик. Если новая метрика может содержать уникальных пользователей, которые не учтены другими метриками, то нужно создать функцию, которая будет принимать данные метрики и возвращать массив никнеймов найденных пользователей. Далее эту функцию нужно вставить в switch-case из “admin/metric/src/entities/Metric/instances/extractUsers.ts”, например:

case "Commits":

return extractCommitsUsers(metric.data);

1. Добавить UI метрики.

7.1 Создать .tsx-файл в папке из шага 2 и определить функциональный React-компонент с пропсами:

interface TotalCommitsProps extends TotalCommitsMetric {

dependencies: IMetric[];

className?: string;

}

const TotalCommits: FC<TotalCommitsProps> = ({

className,

dependencies,

...metric

}) => {

Где “TotalCommits” заменить на название метрики. Интерфейс из первой строки берётся из шага 3.

7.2 Добавить созданный React-компонент в switch-case в файле “metric/admin/src/entities/Metric/instances/ui.tsx”, например:

case MetricName.TotalCommits:

return <TotalCommits {...metric} dependencies={dependencies} />;

1. Добавить фильтрацию данных по календарю и списку пользователей (для всех добавленных на данный момент метрик это файл hooks/useFilter.tsx)  
   Хорошо будет ещё в файле “index.ts” папки из шага 2 именным экспортом экспортировать этот хук, чтобы им можно было пользоваться и в других метриках, которые будут использовать данную метрику как зависимость.  
     
   Для фильтрации по календарю вызвать хук:

const calendar = useAppSelector((state) => state.metric.calendar);

сalendar - объект со свойствами start (выбранная дата начала) (string | null) и end (выбранная дата конца) (string | null)

Для фильтрации по пользователям вызвать хук:

const users = useSelectedUsers(resourceId);

users - массив строк (выбранных никнеймов пользователей)