



MILCT
DEV

Product Baseline

Gruppo MILCTdev — Progetto OpenAPM
milctdev.team@gmail.com

Versione	2.0.0
Redazione	Mattia Bano Isacco Maculan
Verifica	Carlo Munarini
Approvazione	Isacco Maculan
Uso	Esterno
Distribuzione	Kirey Group Prof. Tullio Vardanega Prof. Riccardo Cardin Gruppo MILCTdev

Descrizione

Questo documento descrive l'architettura dell'applicazione realizzata per il progetto OpenAPM.

Registro delle modifiche

Versione	Ruolo	Nominativo	Descrizione	Data
2.0.0	Responsabile	Isacco Maculan	Approvazione del documento per il rilascio	2018-06-10
1.1.0	Verificatore	Carlo Munarini	Verifica documento	2018-06-05
1.0.4	Progettista	Isacco Maculan	Aggiunti riferimenti normativi e informativi	2018-05-20
1.0.3	Progettista	Mattia Bano	Aggiunti termini di glossario	2018-05-20
1.0.2	Progettista	Isacco Maculan	Stesura §1	2018-05-20
1.0.1	Progettista	Mattia Bano	Aggiunto changelog	2018-05-20
1.0.0	Responsabile	Isacco Maculan	Approvazione del documento per il rilascio	2018-05-10
0.3.0	Verificatore	Tommaso Carraro	Verifica documento	2018-05-10
0.2.1	Progettista	Luca Dal Medico	Modifiche a seguito di colloquio Agile	2018-05-10
0.2.0	Verificatore	Isacco Maculan	Verifica documento	2018-04-22
0.1.2	Progettista	Leonardo Nodari	Stesura §5	2018-04-17
0.1.1	Progettista	Leonardo Nodari	Stesura §4	2018-04-17
0.1.0	Verificatore	Isacco Maculan	Verifica documento	2018-04-12
0.0.3	Progettista	Luca Dal Medico	Stesura §3	2018-04-10
0.0.2	Progettista	Luca Dal Medico	Stesura §2	2018-04-05
0.0.1	Progettista	Luca Dal Medico	Inserimento template documento	2018-04-05

Indice

1 Introduzione	5
1.1 Scopo del documento	5
1.2 Scopo del prodotto	5
1.3 Premessa	5
1.4 Riferimenti	5
1.4.1 Riferimenti normativi	5
1.4.2 Riferimenti informativi	5
1.5 Glossario	6
2 Architettura generale	7
3 Diagrammi delle classi	9
3.1 Classe SchedulingConfigurer	9
3.2 Package dispatchers	10
3.3 Package jobs	11
3.4 Diagramma Metric - Alert	12
3.5 Sotto-package evaluators	13
3.6 Sottopackage verifiers	14
3.7 Diagramma Alerts - Actions	15
3.8 Package actions	16
3.9 Package databases	17
3.10 Package operators	18
3.11 Package strategies	19
4 Design pattern	20
5 Diagrammi di sequenza	21
5.1 Diagramma di generazione metrica	21
5.2 Diagramma MetricCreated - AlertsManager	22
5.3 Diagramma AlertTriggered - ActionsManager	23

Immagini

1	Architettura generale della applicazione	7
2	Diagramma della classe SchedulingConfigurer	9
3	Diagramma del package dispatchers	10
4	Diagramma di jobs - metric	11
5	Diagramma Metric - Alert	12
6	Diagramma del package alerts.evaluators	13
7	Diagramma del sottopackage verifiers	14
8	Diagramma alert - actions	15
9	Diagramma del package actions	16
10	Diagramma del package databases	17
11	Diagramma del package operators	18
12	Diagramma del package strategies	19
13	Diagramma di sequenza di generazione metrica	21
14	Diagramma di sequenza MetricCreated - AlertsManager	22
15	Diagramma di sequenza AlertTriggered - ActionsManager	23

1 Introduzione

1.1 Scopo del documento

Il seguente documento ha lo scopo di illustrare e descrivere l'architettura progettata per l'applicazione realizzata durante il *progetto_G* OpenAPM.

Descrivendo l'architettura generale e andando in dettaglio spiegando classi e *design pattern_G* utilizzati, il lettore potrà avere una visione d'insieme delle componenti dell'applicazione, avendo quindi una maggiore consapevolezza sulle scelte adottate dal *team_G*.

1.2 Scopo del prodotto

Lo scopo del *prodotto_G* è realizzare un set di funzioni basate su *Elasticsearch_G* e *Kibana_G* per interpretare i dati raccolti da un *Agent_G*. I dati interpretati forniranno a *DevOps_G* statistiche e informazioni utili per comprendere il funzionamento della propria applicazione. In particolare si richiede lo sviluppo di un motore di generazione di *metriche_G* da *trace_G*, un motore di generazione di *baseline_G* basato sulle metriche del punto precedente, e un motore di gestione di *critical event_G*.

1.3 Premessa

Data la complessità dell'architettura, alcuni diagrammi possono richiedere uno zoom per essere visionati correttamente.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- **Norme di progetto:** *Norme di Progetto v4.0.0*;

1.4.2 Riferimenti informativi

Spring Batch

<https://projects.spring.io/spring-batch/>
(ultima consultazione effettuata in data 2018-06-05);

Spring e-mail

<http://www.baeldung.com/spring-email>
(ultima consultazione effettuata in data 2018-06-05);

ElasticSearch

<https://www.elastic.co/products/elasticsearch>
(ultima consultazione effettuata in data 2018-06-05);

Spring
<https://spring.io/>
(ultima consultazione effettuata in data 2018-06-05).

1.5 Glossario

All'interno del documento sono presenti termini che possono assumere significati diversi a seconda del contesto. Per evitare ambiguità, i significati dei termini complessi adottati nella stesura della documentazione sono contenuti nel documento *Glossario v3.0.0*. Per segnalare un termine del testo presente all'interno del Glossario verrà aggiunta una _G a pedice e il testo sarà in corsivo.

2 Architettura generale

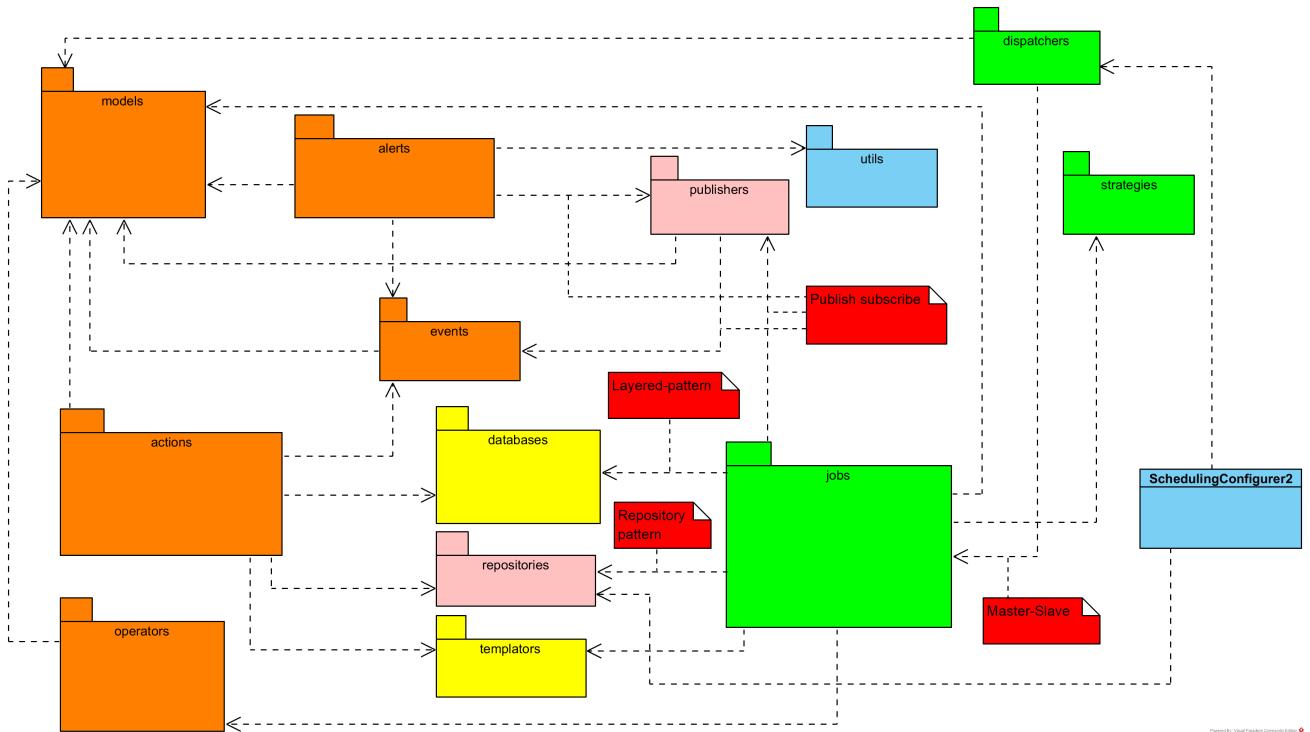


Figura 1: Architettura generale dell'applicazione

L'immagine raffigura il diagramma dei package che descrive l'architettura generale dell'applicazione. L'applicativo è composto dalle seguenti parti:

- **Scheduler Configurer**: classe che si occupa di configurare lo scheduler di Spring. Si occupa della creazione dei dispatchers utili al funzionamento dell'applicazione;
- **actions**: package che contiene tutte le classi che rappresentano azioni di rimedio. Le azioni di rimedio vengono eseguite nel caso in cui si verifichi un alert;
- **alerts**: package che contiene le classi che descrivono lo scatenarsi di un evento in cui è necessario eseguire un'azione di rimedio;
- **databases**: package che contiene classi necessarie alla raccolta di dati utili durante l'esecuzione;
- **dispatchers**: package che contiene le classi che permettono la creazione di dispatchers per la richiesta di generazione di metriche e baseline;
- **events**: package che contiene le classi che rappresentano il manifestarsi di un evento;
- **jobs**: package che contiene le classi che rappresentano le operazioni che può svolgere l'applicazione;
- **models**: package che contiene le classi che rappresentano tutte le configurazioni per le componenti dell'applicazione;
- **operators**: package che raccoglie tutti gli operatori coinvolti nei diversi calcoli presenti nell'applicazione;

- **publishers:** package che contiene le classi che si occupano di generare notifiche dell'avvenimento di un evento (metrica generata, alert generato);
- **repositories:** package che contiene le classi necessarie per prelevare le configurazioni degli oggetti da Elasticsearch;
- **strategies:** package che contiene le classi che descrivono le strategie con cui si recuperano le baseline;
- **templators:** package che contiene le classi utili a creare date e operatori;
- **utils:** package che contiene classi di utilità.

3 Diagrammi delle classi

3.1 Classe SchedulingConfigurer

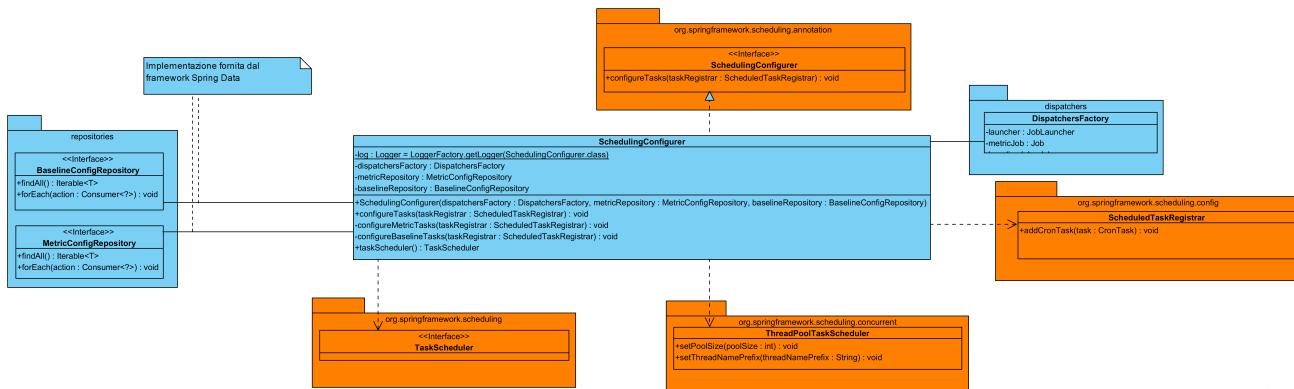


Figura 2: Diagramma della classe SchedulingConfigurer

La classe SchedulingConfigurer:

- costruisce un oggetto *DispatchersFactory* necessario per la costruzione di dispatchers;
- costruisce un oggetto *MetricConfigRepository* necessario per andare a prelevare da Elasticsearch la corretta configurazione per le metriche da creare;
- costruisce un oggetto *BaselineConfigRepository* necessario per andare a prelevare da Elasticsearch la corretta configurazione per le baseline da creare;
- definire un oggetto *TaskScheduler* necessario per la gestione dei *task* riguardanti le metriche e le baseline all'interno dell'applicazione.

La classe implementa l'interfaccia *SchedulerConfigurer* del framework Spring. Questo perché permette di creare un oggetto *TaskScheduler* capace di aggiungere operazioni alla coda delle operazioni da svolgere. La classe utilizza le seguenti classi/interfacce:

- **DispatchersFactory**: grazie a questa classe è possibile creare dispatchers che gestiscano la coda dei processi;
- **BaselineConfigRepositories**: permette di prelevare da Elasticsearch la configurazione prevista per la costruzione di una baseline;
- **MetricConfigRepositories**: permette di prelevare da Elasticsearch la configurazione prevista per la costruzione di una metrica;
- **TaskScheduler**: questa classe permette la programmazione di oggetti *Runnables* in base a diversi tipi di triggers;
- **ThreadPoolTaskScheduler**: questa classe è l'implementazione dell'interfaccia *TaskScheduler* utilizzata in OpenAPM;
- **SchedulerTaskRegistrar**: è utilizzata come classe di supporto nella registrazione di operazioni nel *TaskScheduler*.

3.2 Package dispatchers

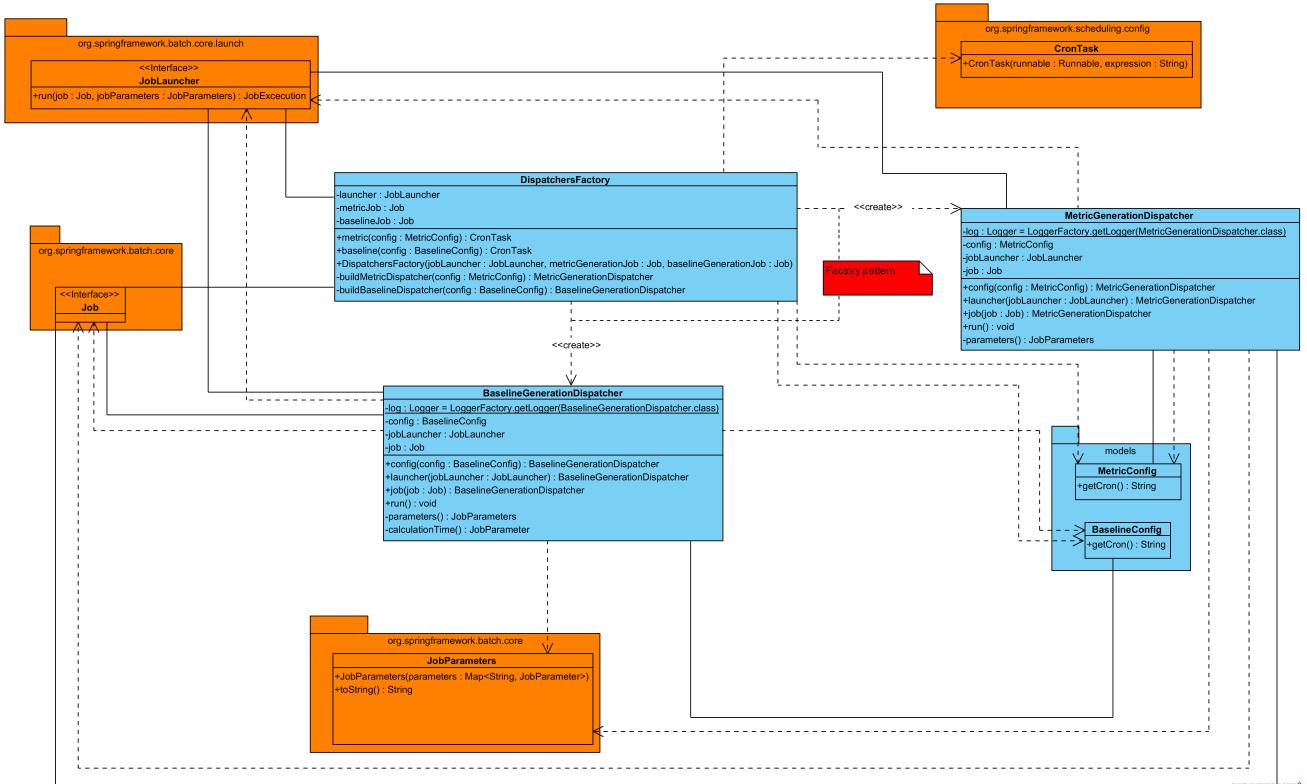


Figura 3: Diagramma del package dispatchers

Il diagramma rappresenta il package dispatchers, composto dalle seguenti classi:

- **DispatchersFactory**: questa classe permette la creazione di dispatchers in base ad una configurazione data in input lasciando alle sotto-classi l'implementazione effettiva, si basa infatti sul factory pattern;
- **BaselineGenerationDispatcher**: questa classe rappresenta un dispatcher per la generazione di baseline basate sulle metriche prodotte nell'ultima ora;
- **MetricGenerationDispatcher**: questa classe rappresenta un dispatcher per la generazione di metriche basante sulle traces raccolte dall'ora attuale fino ad un tempo indicato dalla strategia scelta.

3.3 Package jobs

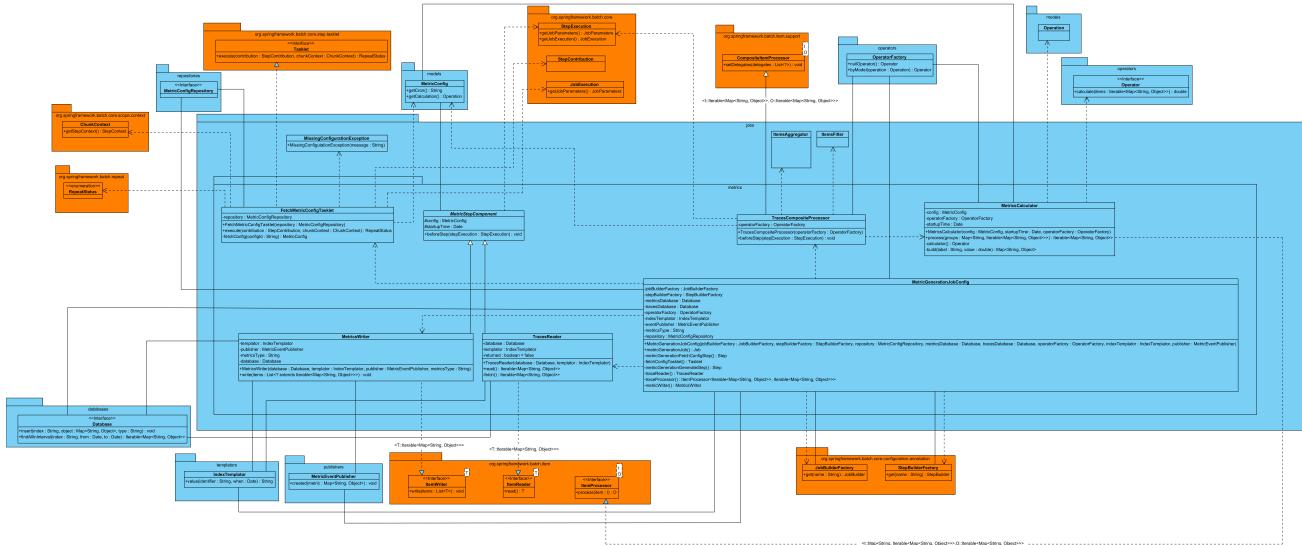


Figura 4: Diagramma della parte sulle metriche del package jobs

Il diagramma rappresenta il compito di generare una metrica. Vi è un diagramma equivalente per le baseline, non presentato perché simile. Il package jobs, per la parte dedicata alle metriche, è composto dalle seguenti classi:

- **FetchMetricConfigTasklet**: classe che preleva dal database la configurazione per la creazione delle metriche basandosi su un identificatore; essa può lanciare una *MissingConfigurationException* in caso la configurazione sia presente nel database;
- **MetricStepComponent**: classe astratta che permette di prelevare dei parametri dal contesto di un lavoro prima dell'esecuzione dello stesso;
- **MetricsWriter**: implementazione di *MetricStepComponent* che permette di salvare la metriche prodotte nel database;
- **TracesReader**: implementazione di *MetricStepComponent* che permette di leggere le traces presenti nel database;
- **TracesCompositeProcessor**: questa classe definisce una serie di processori coinvolti nella generazione di metriche a partire da traces, questa classe è necessaria per sapere se è presente un evento di tipo *BeforeStep*;
- **MetricGenerationJobConfig**: questa classe rappresenta la configurazione per l'operazione di creazione di metriche a partire da traces;
- **MetricsCalculator**: questa classe permette il calcolo di una metrica a partire da un insieme di traces in input;
- **ItemsAggregator**: classe che aggrega tutti gli input suddivisi in gruppi in base ad una configurazione data;
- **ItemsFilter**: classe che filtra tutti gli input in base ad una configurazione data;
- **MissingConfigurationException**: eccezione che indica che il compito indicato in input non è presente nel database.

3.4 Diagramma Metric - Alert

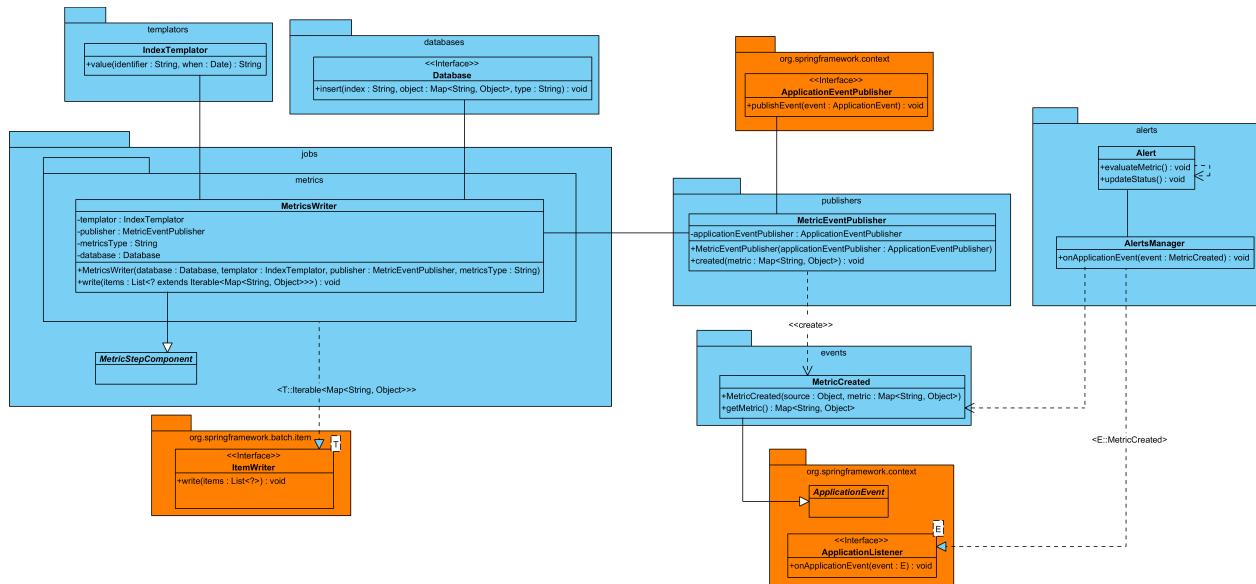


Figura 5: Diagramma che mostra l'interazione tra il package metric e il package alert

Il diagramma rappresenta l'interazione tra il package metric e il package alert.

Al momento della creazione di una metrica viene lanciato un evento MetricCreated e vengono controllate le condizioni degli alert per tale metrica.

Le classi più importanti sono:

- **MetricsWriter**: essa scrive nel database una nuova metrica con il metodo `void write(List<? extends Iterable<Map<String, Object>>)` utilizzando un *IndexTemplate* per il nuovo indice;
 - **MetricEventPublisher**: una volta creata una nuova metrica, essa notifica questo evento
 - **MetricCreated**: rappresenta l'evento di creazione di una metrica ed è un'estensione di un *ApplicationEvent*, questo per far sì che un *ApplicationListener*, nel prodotto esteso dalla classe *AlertsManager*, possa rimanere in ascolto di tutti gli eventi accaduti;
 - **AlertsManager**: gestisce gli alert dei prodotti in base agli eventi accaduti, rimanendo in ascolto di oggetti *MetricCreated*, controllandole successivamente gli alerts necessari.

3.5 Sotto-package evaluators

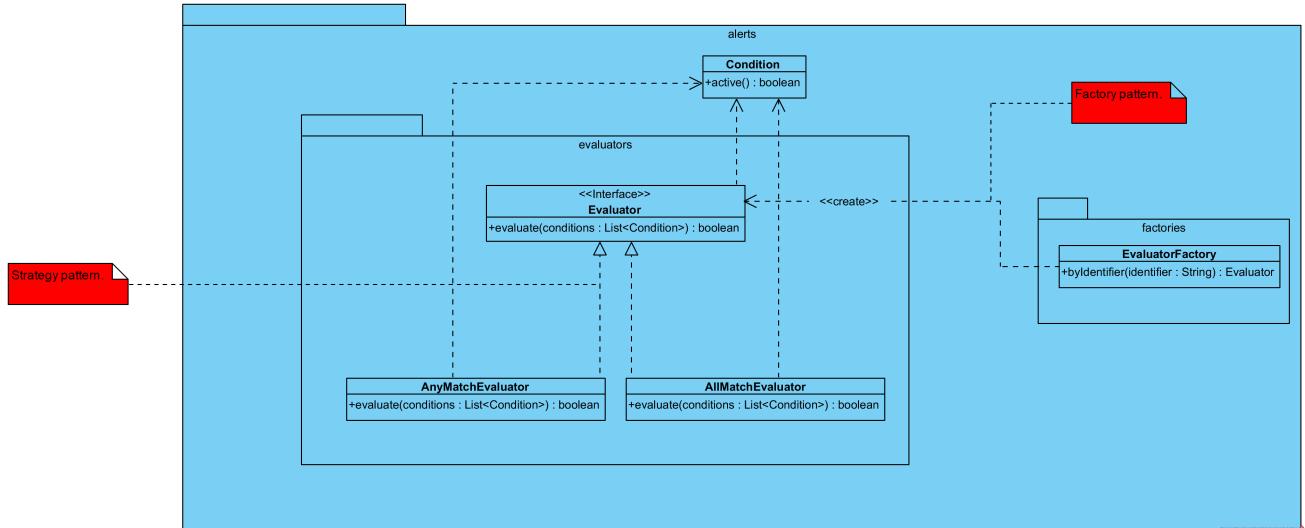


Figura 6: Diagramma del package alerts.evaluators

Il diagramma rappresenta il sotto-package evaluators del package alerts. Un evaluator controlla se le conditions per l'alert sono soddisfatte, generando in caso un critical event.

Vengono di seguito spiegate alcune classi ed interfacce:

- **Evaluator**: interfaccia che rappresenta un oggetto il quale riceve una serie di condizioni in input e, a seconda della politica adottata, decide se è necessario lanciare un'azione di rimedio. Anche in queste classi, come in *Action*, è stato scelto di utilizzare lo strategy pattern per separare l'implementazione della politica di valutazione con il client che lo utilizza;
- **Condition**: questa classe rappresenta una condizione che, se vera, indica la necessità del lancio di un'azione di rimedio. Una lista di queste condizioni viene analizzata da un *Evaluator*;
- **EvaluatorFactory**: questa classe permette di costruire un valutatore basandosi su una configurazione data; per far ciò si è scelto di utilizzare il factory pattern, come in *ActionFactory*, per dare la responsabilità dell'implementazione del valutatore corretto alle sue sotto-classi.

3.6 Sottopackage verifiers

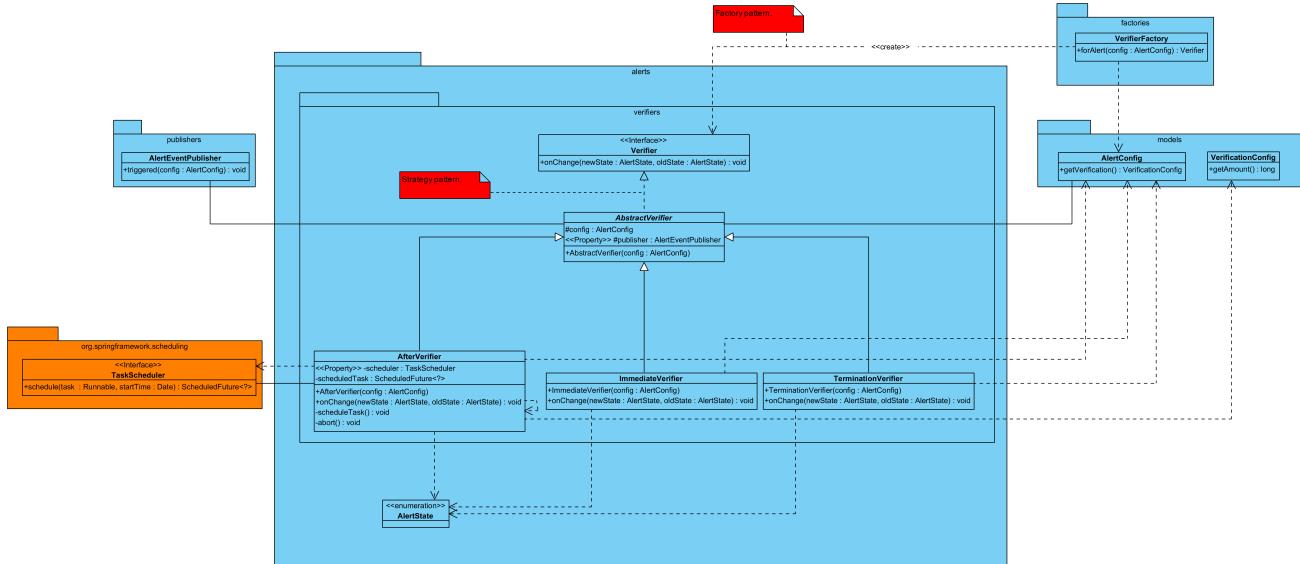


Figura 7: Diagramma del sottopackage verifiers

Il diagramma rappresenta il package `verifiers`, sottopackage di `alerts`. I verificatori servono a porre una condizione sul lancio di un'azione di rimedio, infatti questa classe riceve in input uno stato per un alert e decide se o quando lanciare un'azione di rimedio.

Il package è composto dalle seguenti classi/interfacce:

- **Verifier**: interfaccia per un verificatore, questa astrazione permette di separare il client che utilizza l'oggetto dall'implementazione effettiva di questo, seguendo l'organizzazione delle classi del pattern strategy;
- **AbstractVerifier**: classe astratta per un verificatore;
- **AfterVerifier**: verificatore che lancia un'azione di rimedio dopo alcuni secondi il verificarsi del cambio di stato dell'alert;
- **ImmediateVerifier**: verificatore che lancia un'azione di rimedio non appena si verifica un cambio di stato dell'alert;
- **TerminationVerifier**: verificatore che lancia un'azione di rimedio dopo che l'evento scatenante l'alert sia terminato;
- **AlertState**: classe che descrive tutti i possibili stati di un alert;
- **VerifierFactory**: classe che permette di costruire un verificatore in base ad una configurazione scelta, è stato scelto di organizzare l'architettura di questa classe seguendo il factory pattern, già descritto precedentemente;

3.7 Diagramma Alerts - Actions

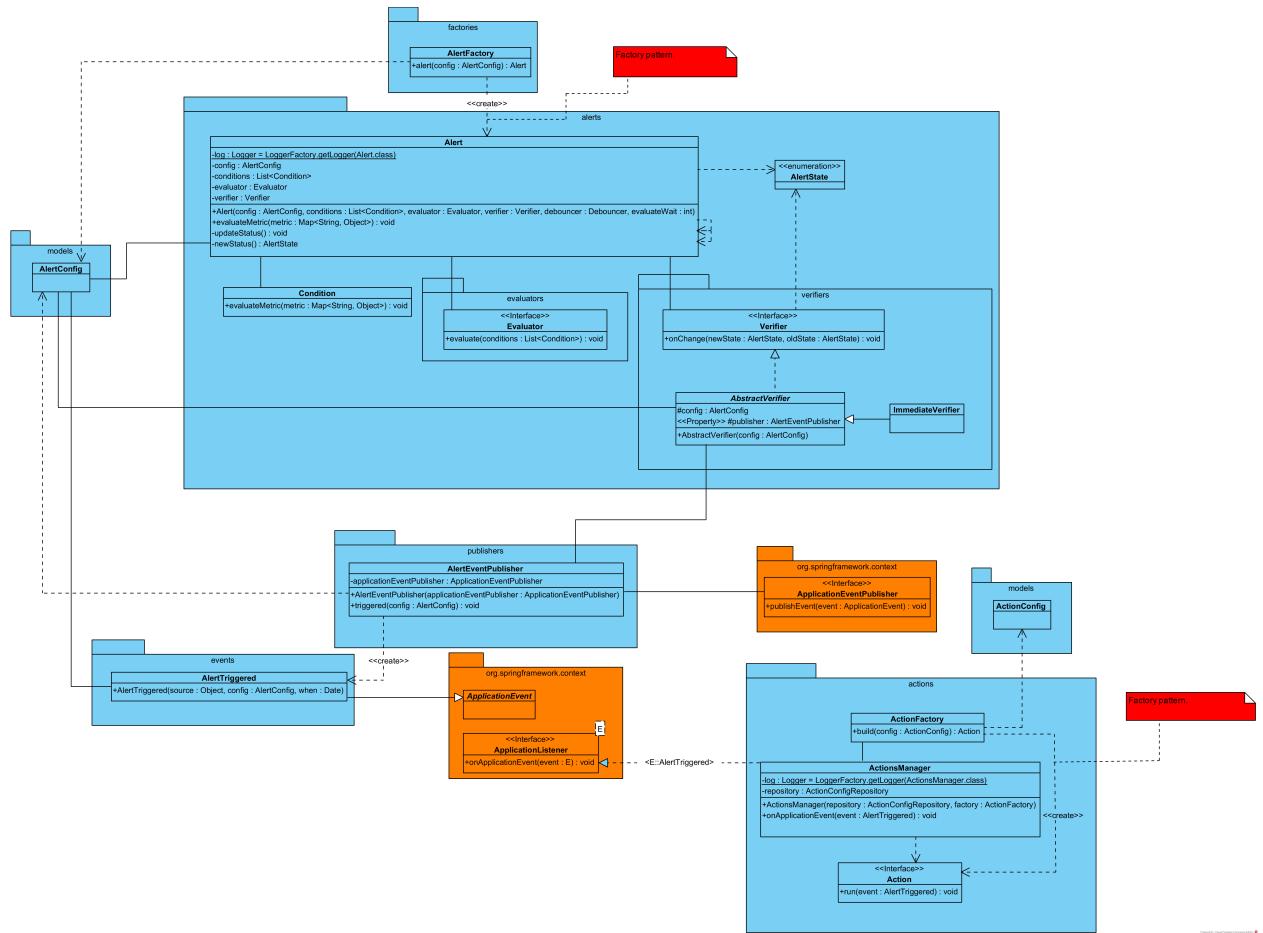


Figura 8: Diagramma che mostra l'interazione tra alerts e actions

Il diagramma di classe mostra le dipendenze tra il package alerts e il package actions. Si è deciso di creare tale diagramma per comprendere quali classi agiscono nell'interazione tra i due packages.

Interazioni:

- **Alert - Verifier**: l'oggetto *Verifier*, alla notifica di un evento, andrà a modificare *AlertStatus* secondo la sua politica attraverso il metodo `void onChange(AlertState, AlertState)`, causando così lo scattare di un *Alert*;
- **AbstractVerifier - AlertEventPublisher**: l'oggetto *AlertEventPublisher* è responsabile della notifica ad un *AbstractVerifier* del verificarsi di un evento, quest'ultimo, o meglio l'implementazione scelta di quest'ultimo, va a decidere se e quando lanciare un'azione di rimedio;
- **AlertEventPublisher - AlertTriggered**: allo scattare di un alert, *AlertEventPublisher* crea un oggetto *AlertTriggered* che rappresenta l'evento di un alert scattato;
- **ActionsManager - AlertTriggered**: allo scattare di un alert, quindi alla creazione di un *AlertTriggered*, l'oggetto di tipo *ActionsManager*, che rimane in ascolto di eventi di tipo *AlertTriggered* tramite *ApplicationListener*, gestisce le azioni di rimedio da utilizzare.

3.8 Package actions

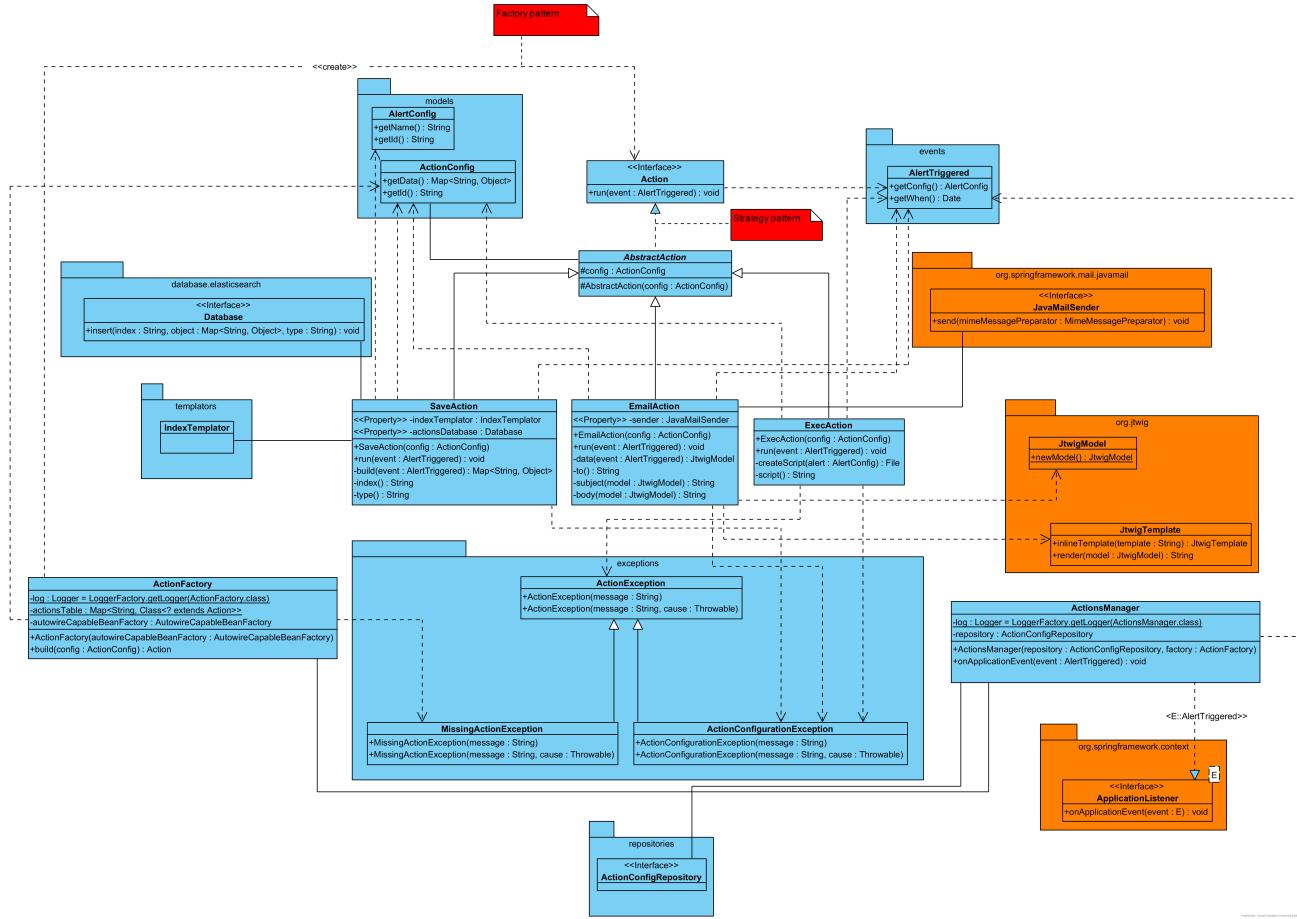


Figura 9: Diagramma del package actions

Il diagramma di classe rappresenta il package actions, diviso nelle seguenti parti:

- **Action:** interfaccia rappresentante un'azione di rimedio, nella costruzione di queste classi si è deciso di utilizzare il pattern strategy, cioè creando un'interfaccia che non implementi le azioni da svolgere e lasciando questo compito alle sotto-classi. Questo permette rendere indipendente l'algoritmo utilizzato nell'azione di rimedio dal client che la utilizza;
- **ActionException:** questa classe permette di gestire anomalie nell'esecuzione e di lanciare un'eccezione nel caso un'azione di rimedio non possa essere eseguita, ad esempio se la configurazione desiderata per l'azione non è corretta o se l'azione richiesta non esiste;
- **ActionFactory:** questa classe è responsabile della creazione ed utilizza il factory pattern, cioè permette di creare un astrazione nella creazione delle classi definendo un oggetto astratto e lasciando alle sotto-classi la definizione dell'oggetto;
- **ActionsManager:** questa classe gestisce le azioni di rimedio nel caso scatti un alert.

3.9 Package databases

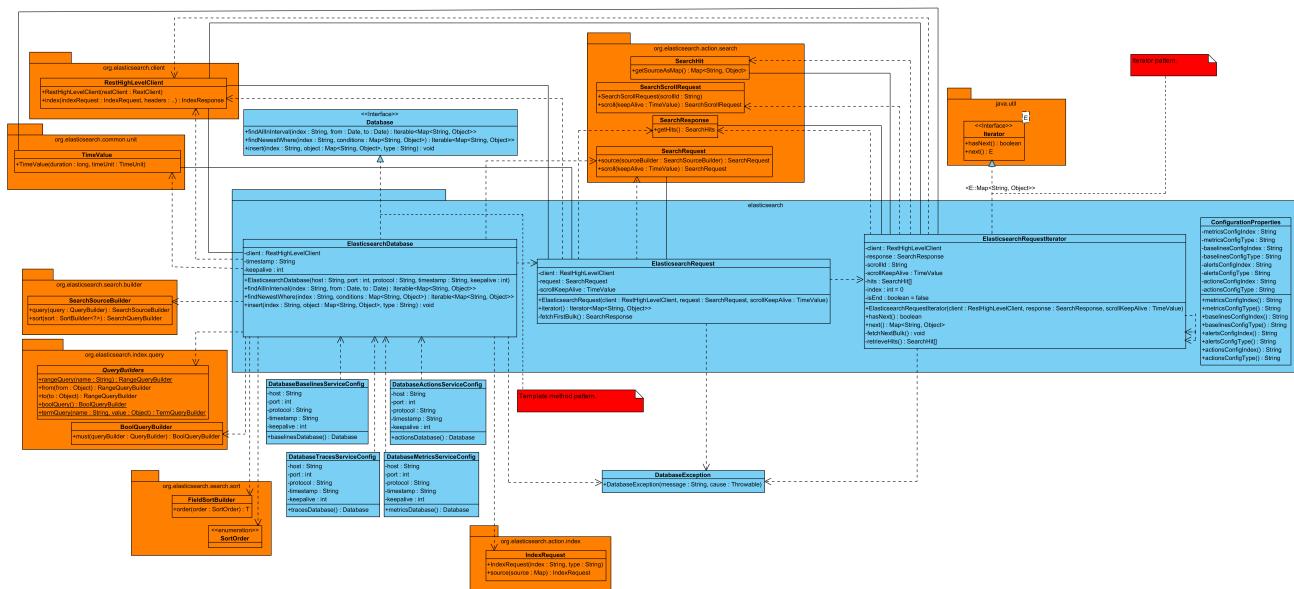


Figura 10: Diagramma del package databases

Il diagramma rappresenta il package databases contenente le classi per gestire dati da memorizzare durante l'esecuzione di OpenAPM.

Vengono spiegate le seguenti classi/interfacce:

- **Databases:** questa interfaccia rappresenta in modo astratto in database necessario alla raccolta di dati utili nell'esecuzione. MILCTdev ha voluto sviluppare questa classe utilizzando il *template* method pattern, cioè creare una classe astratta e lasciare l'implementazione dei metodi alle varie sotto-classi, questo ha reso estendibile il prodotto ad ulteriori tipi di database, infatti in OpenAPM è presente l'implementazione di questa classe per usare un database con Elasticsearch con la classe *ElasticsearchDatabase*;
 - **ElasticSearchIterator:** questa classe permette lo scorrimento dei risultati ottenuti da una richiesta ad un *ElasticsearchDatabase*, per implementarlo MILCTdev ha seguito l'iterator pattern, creando così un iteratore che tenga traccia della posizione corrente e calcoli il prossimo elemento così da poter attraversare tutti gli elementi ottenuti dal database;
 - **ConfigurationProperties:** viene utilizzata questa classe per configurare i metadati del container IoC di Spring anziché farlo tramite XML, in particolare le proprietà descritte nel file “*application.properties*” vengono legate ad un corrispettivo Bean affinché le si possano risolvere nelle annotazioni quando si voglia far uso di Dependency Injection.

3.10 Package operators

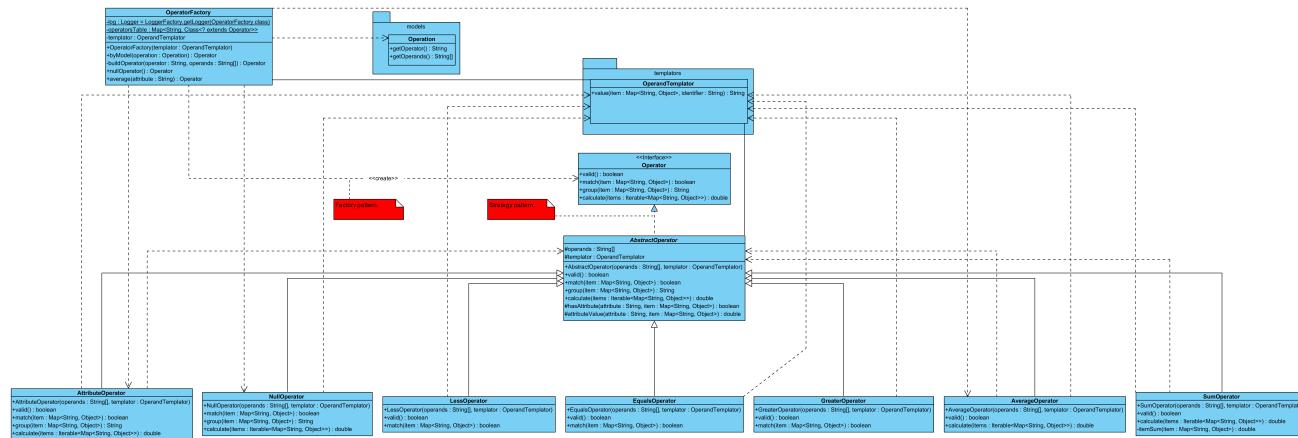


Figura 11: Diagramma del package operators

Il diagramma rappresenta il package operators. In tale package sono contenute tutte le classi che rappresentano gli operatori che vengono utilizzati nelle operazioni di OpenAPM.
Spiegazione:

- **Operator**: questa interfaccia rappresenta operatori in grado di eseguire operazioni su oggetti, MILCTdev ha scelto di basarsi sullo strategy pattern, descritto precedentemente, per lo sviluppo di questa classe e delle sue sotto classi, per rendere maggiormente estendibile il prodotto;
 - **AbstractOperator**: classe astratta con un’implementazione base di un operatore;
 - **AttributeOperator**: operatore per ottenere l’attributo di un oggetto;
 - **NullOperator**: operatore nullo che non effettua nessuna operazione o calcolo;
 - **LessOperator**: operatore per verificare che il primo operatore sia minore del secondo;
 - **EqualsOperator**: operatore per verificare che due valori siano uguali;
 - **GreaterOperator**: operatore per verificare che il primo operatore sia maggiore del secondo;
 - **AverageOperator**: operatore per calcolare la media di un attributo in un insieme di traces;
 - **SumOperator**: operatore che effettua la somma di un attributo in un insieme di oggetti.
 - **OperatorFactory**: classe che permette di costruire un operatore in base ad una configurazione scelta, MILCTdev ha scelto di organizzare l’architettura di questa classe seguendo il factory pattern, già descritto precedentemente;
 - **OperandTemplator**: classe utilizzata per gestire gli operandi nelle operazioni.

3.11 Package strategies

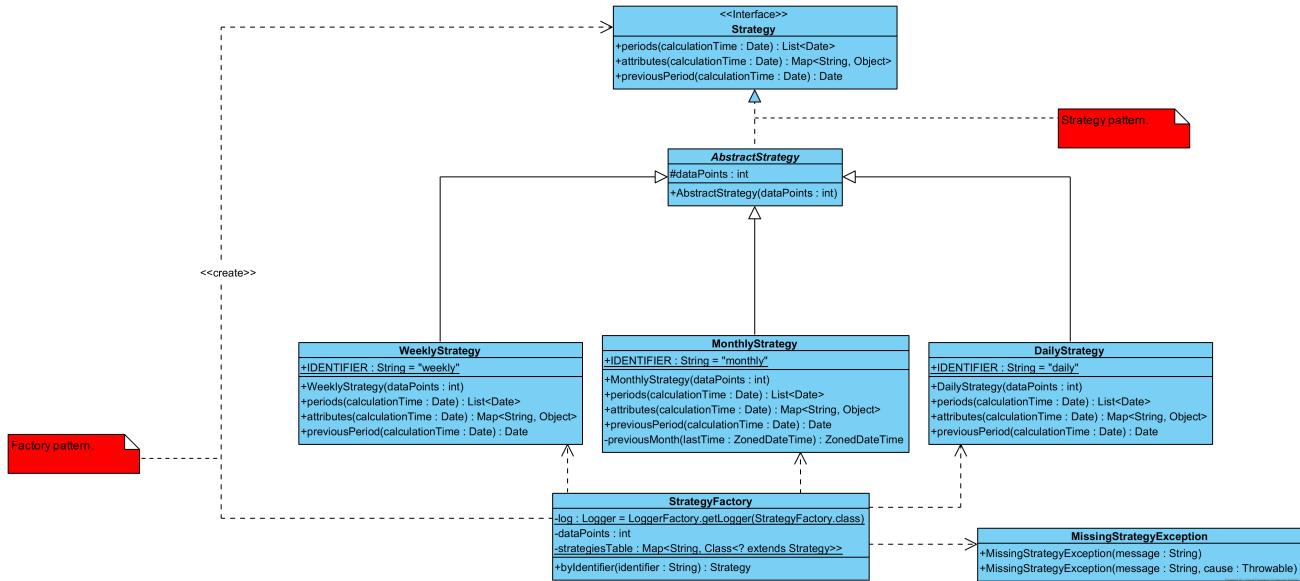


Figura 12: Diagramma del package strategies

Il diagramma rappresenta il package strategies composto dalle seguenti classi:

- **Strategy**: interfaccia di una strategia per il calcolo di una baseline, per rendere indipendente la strategia scelta del client che la utilizza è stato scelto di organizzare queste classi basandosi sullo strategy pattern già visto precedentemente anche in altre classi;
- **AbstractStrategy**: classe astratta di una strategia che gestisce le proprietà di default;
- **WeeklyStrategy**: classe per calcolare baseline su base settimanale;
- **MonthlyStrategy**: classe per calcolare baseline su base mensile;
- **DailyStrategy**: classe per calcolare baseline su base giornaliera;
- **StrategyFactory**: classe che permette di costruire una strategia in base ad una configurazione scelta, questa classe si basa sul factory pattern per rendere indipendente l'oggetto utilizzato della sua effettiva implementazione;
- **MissingStrategyException**: eccezione lanciata quando non è possibile definire una strategia a causa di parametri in input non validi.

4 Design pattern

Nell'architettura sono presenti altri pattern oltre a quelli già analizzati. Sono stati utilizzati i seguenti design pattern di Spring:

- **Repository pattern:** fornito da Spring Data, utilizzato per prelevare le configurazioni dal database;
- **Builder pattern:** fornito da *Spring Batch*, utilizzato per configurare i Job di generazione metriche / baseline;
- **Singleton pattern:** utilizzato per tutte le factory, templator e più in generale ogni volta viene dichiarato un Bean.

5 Diagrammi di sequenza

5.1 Diagramma di generazione metrica

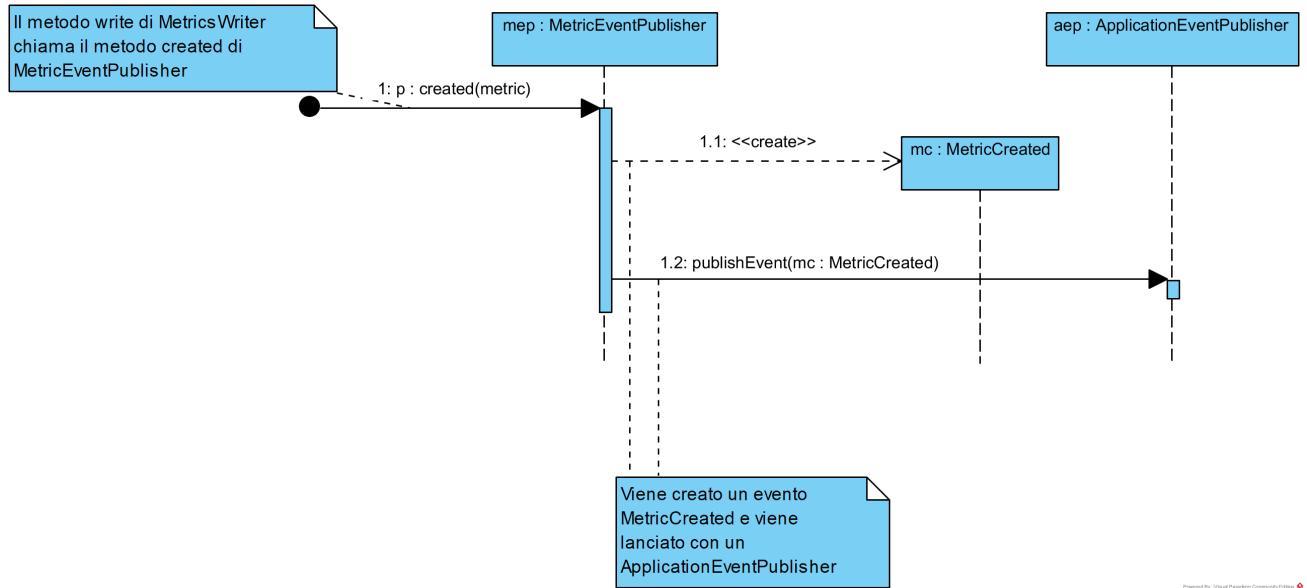


Figura 13: Diagramma di sequenza che mostra le operazioni per generare una metrica

Il diagramma di sequenza mostra le operazioni per la generazione di una metrica. Il flusso è il seguente:

1. viene notificato il `MetricEventPublisher` della creazione di una metrica tramite il metodo `created`;
2. `MetricEventPublisher` si occupa di creare un evento `MetricCreated` e di lanciarlo tramite un'`ApplicationEventPublisher`.

5.2 Diagramma MetricCreated - AlertsManager

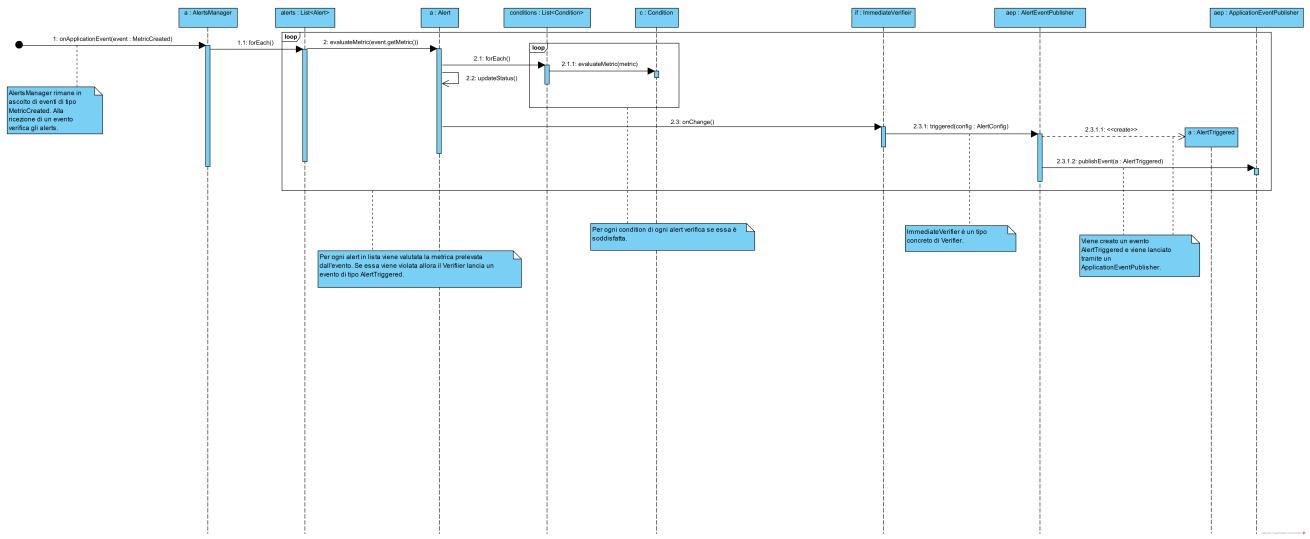


Figura 14: Diagramma di sequenza delle operazioni di AlertsManager

Il diagramma di sequenza rappresenta le operazioni della classe AlertsManager. Il flusso è il seguente:

1. AlertsManager rimane in ascolto di un evento MetricCreated;
2. per ogni alert in lista viene valutata la metrica prelevata dall'evento;
3. la valutazione avviene tramite valutazione di tutte le conditions di ogni alert;
4. in caso di criticità un verificatore invocherà il metodo triggered dell'AlertEventPublisher;
5. AlertEventPublisher si occupa di creare un evento AlertTriggered e di lanciarlo con un ApplicationEventPublisher.

5.3 Diagramma AlertTriggered - ActionsManager

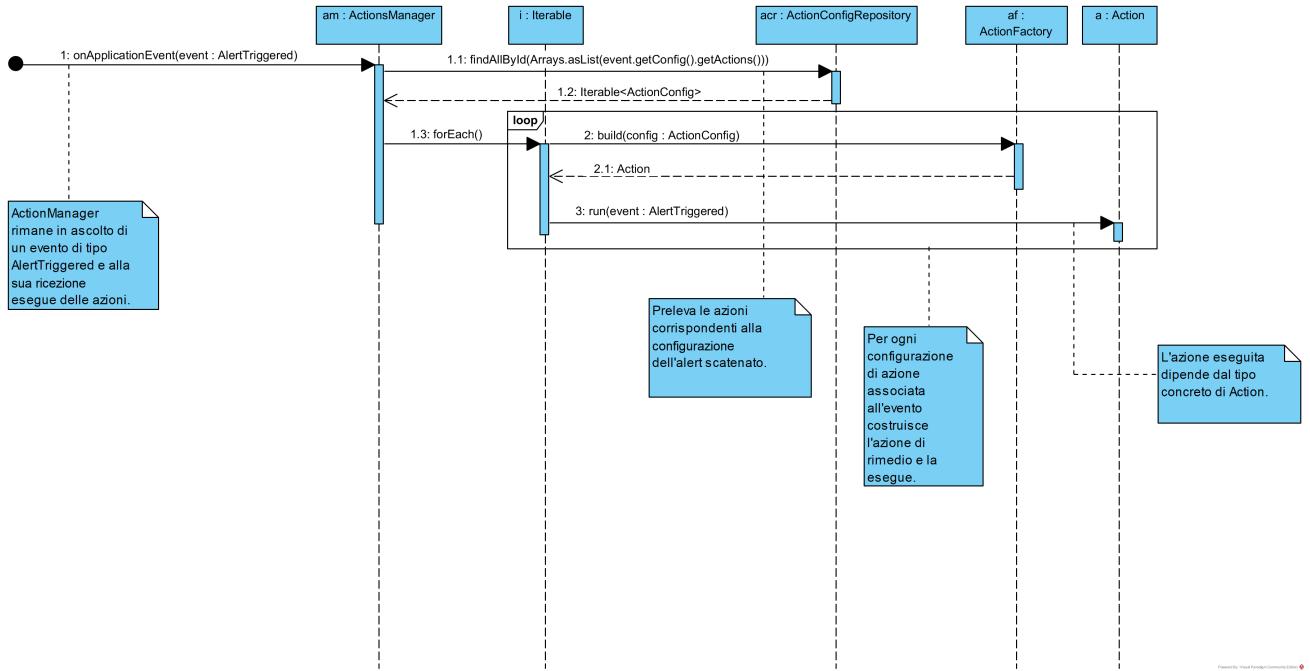


Figura 15: Diagramma di sequenza delle operazioni di ActionsManager

Il diagramma di sequenza rappresenta le operazioni della classe ActionsManager. Il flusso è il seguente:

1. ActionsManager rimane in ascolto di un evento AlertTriggered;
2. alla ricezione dell'evento vengono raccolte dal database le configurazioni delle azioni associate all'evento;
3. per ogni configurazione di azione, essa viene costruita tramite factory pattern e successivamente eseguita.