Spring-boot

What has changed:

1. Added the essential feature of actuator.
2. Combined two previous features into live reload.
3. Compared difference between spring and spring-boot in both features and identify why the features are essential.
4. Provided detailed explanations about each graph.
5. Reduced unrelated codes in called functions.

Actuator

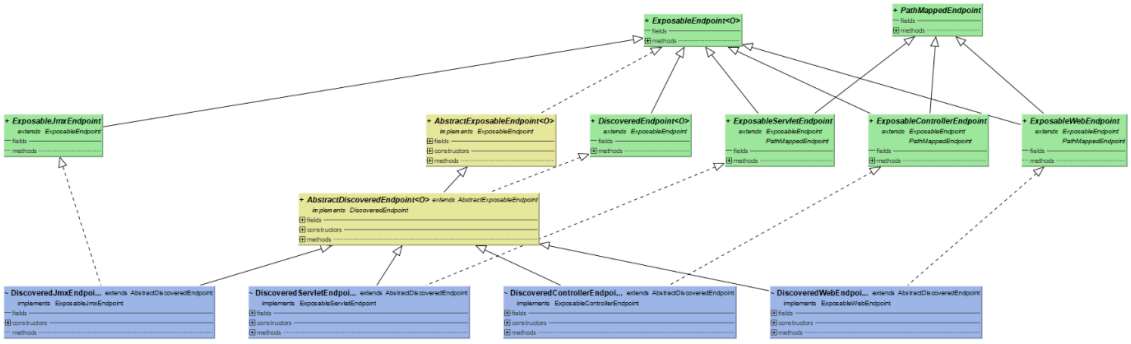
1. Feature Introduction

Actuator is a framework of spring-boot that can monitor and manage system running status in many aspects and gel rid of dealing with cumbersome xml files. More specifically, it provides the functions such as auditing, health status, metrics gathering, etc. Endpoints stand for which part of the system we would like to monitor, and actuator provides varieties of endpoints for user to use. Among those endpoints, health endpoint is a main part of actuator, which denotes the status of specific part.

2. Why Essential

Spring framework is always used to build micro service. Due to this feature, the application developed by spring is always a distributed system, for one function, it could be realized in different machines and parts interact with each other based on interface. So when something went wrong, it's essential to know which part goes wrong and and it's essential to monitor the running status of each part of the application. And even for standalone applications, getting system running status is a fundamental function. To sum up, without actuator, sprint-boot is just spring, it couldn't provide user a easier way to build and monitor applications. The design intention of spring boot is to eliminate the boilerplate configurations required for a spring app, so we believe this is an essential feature.

3. How it Works



Additional graph explanations.

> This graph shows relationship between endpoints. `ExposableEndpoint` is the parent of all the endpoint, and other endpoints who implements the interface. Aside from this, Actuator provides varieties of annotations to accelerate development of endpoint. `HealthEndpoint` utilizes those annotations but annotations can't be shown in the UML graph.

`ExposableEndpoint` is the interface of all the endpoints, it has three functions. `getEndpointId()` will return the id of current endpoint, `isEnableByDefault()` return if current endpoint is enabled, `getOperation()` return all the operations endpoint has.

```
public interface ExposableEndpoint<O extends Operation> {
    EndpointId getEndpointId();

    boolean isEnableByDefault();

    Collection<O> getOperations();

}
```

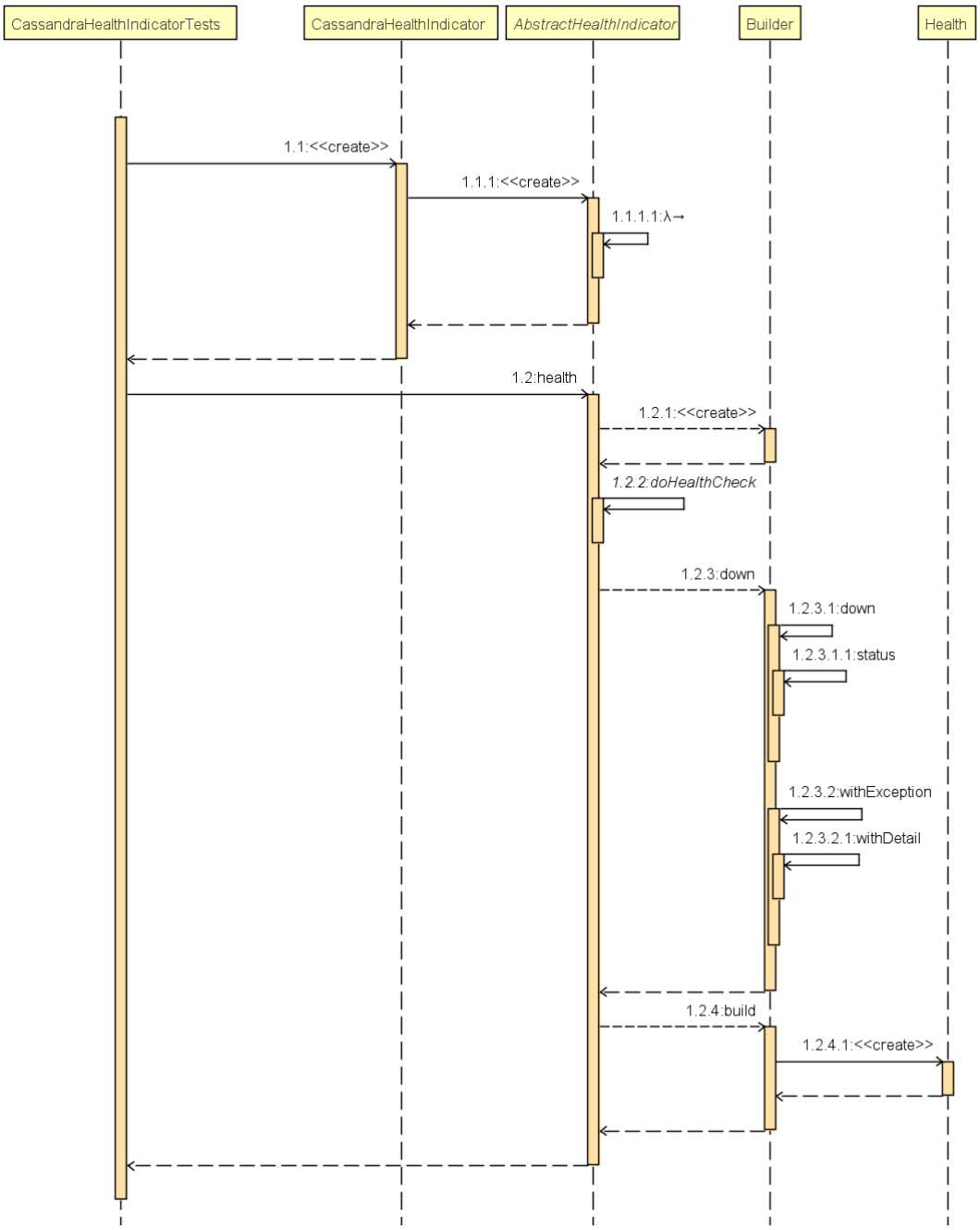
`AbstractExposableEndpoint` is the abstract implementation of `ExposableEndpoint` , and provides the basic realization of the functions. To monitor health statue, we should utilize `HealthEndpoint` , by `getHealth()` function, and in this method, we could find that it is realized by the help of `HealthIndicator`.

```
protected HealthComponent getHealth(HealthContributor contributor, boolean includeDetails) {
        return ((HealthIndicator) contributor).getHealth(includeDetails);
    }
```

`HealthIndicator` is another interface that provides a default method `getHealth()` to provide health information about an endpoint. `AbstractHealthIndicator` is an abstract class for more specific Health Indicator. In this class, it has a key method called `health()` , which is an method of `Health` class. This contains all the information about the current endpoint, and indicator could fetch information from this `Health` object.

`Health` class is the **key** class of health class. It has two important variables, `status` and `details` . This class has a builder design pattern, which means that the variables are set by its inner class `Builder` . Based on different implementations, we have different indicators monitoring different parts of classes.

Take `CassandraHealthIndicator` as an example, the control flow of monitoring its health endpoint is as follows.



Additional graph explanation.

> This simulate the process of checking Cassandra health status. This process starts with creating an indicator. To get health status, we need to call `health()` method. `health()` is a method provided by its parent class `AbstractHealthIndicator`. And then will call `doHealthCheck()` inside `health()` function. If the health passes, it will create a `health` object with `STATUS.UP`, and if not, it will throw an error and caught in the `health()` , and return a `health` object with `STATUS.DOWN`.

In the real application, when we want to monitor the status of Cassandra database, again which kind of health endpoint is trivial, but how the monitor process works is essential, because all the indicator extends the `AbstractHealthIndicator` so their methods are quite similar.

```
    ...
    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        SimpleStatement select = SimpleStatement.newInstance("SELECT
release_version FROM system.local");
        ResultSet results =
this.cassandraOperations.getCqlOperations().queryForResultSet(select);
        if (results.isFullyFetched()) {
            builder.up();
            return;
        }
        String version = results.one().getString(0);
        builder.up().withDetail("version", version);
    }
    ...
```

Firstly, we create a `CassandraHealthIndicator` and it will call the super constructor to set current error message, should error occurs. Secondly, We should call `health()` function to get current health status. In every implementation of indicators, spring-boot doesn't override the `health()` function, but override the `doHealthCheck()` inside the `health()` function.

```
    ...
    @Override
    public final Health health() {
        Health.Builder builder = new Health.Builder();
        try {
            doHealthCheck(builder);
        }
        catch (Exception ex) {
            if (this.logger.isWarnEnabled()) {
                String message = this.healthCheckFailedMessage.apply(ex);
                this.logger.warn(StringUtils.hasText(message) ? message :
DEFAULT_MESSAGE, ex);
            }
            builder.down(ex);
        }
        return builder.build();
    }
    ...
```

In `health()` method, it will create an `Health.Builder` builder, which used to create a health object. If there is an error, the Exception will be caught and for the current builder, its status would be set to `down` . If there isn't an error, the status would be set to `up` . Now we have a builder with set status. Thirdly, it will call `builder.build()` method to create health object. As we have got the health object, we could call `getStatus()` to get current running status.

Live Reload

1. Feature Introduction

Live reload is a unique feature of Sprint boot. When we build web applications with Java, we have to build the whole project and restart server every time to load all our changes. Sprint Boot provide such a solution that it can automatically load all changes without the need of restarting server. This feature is implemented in the package of *spring-boot-devtools*.

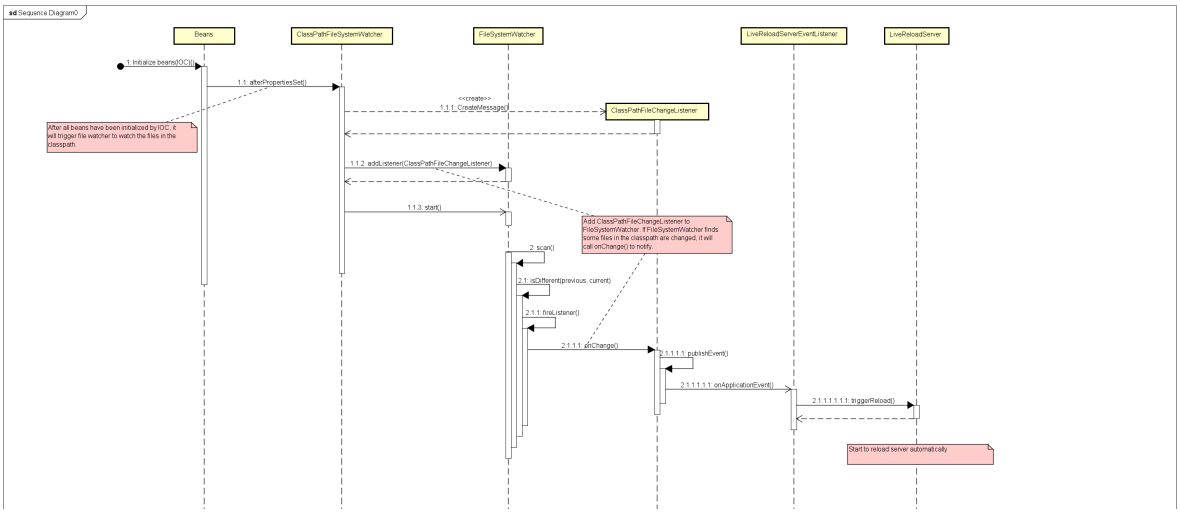
2. Why Essential

Sprint boot is an encapsulation of Spring framework, which provides a lot of features to make developers easier to use. Live reload is an essential feature that enable users saves the cost of building the project and restarting server.

In a typical application development environment, developers need to build the project and deploy/start the application for new changes if they make some changes. If it costs 10 minutes to build project and restart server once, one developer would spend approximately one hour on restarting server everyday. However, by using the feature of live reload, developers could reduce the time spent on verifying the changes they mead, which considerably improve the productivity. Whenever files change in the classpath, applications using *spring-boot-devtools* will cause the application to restart. If it is removed, no one would want to use sprint boot because of the inefficient loading time.

3. How it Works

The sequence of live reload is shown as below and let me explain it with details. I removed some unrelated details and drew the essence of the sequence, and the interaction between different modules.



Sprint boot will monitor files in the classpath and will automatically restart with strategies when files change.

- 1. First, it adds file watcher for files in the classpath.
    - **Class**: *LocalDevToolsAutoConfiguration.java*
    - **Function:** *classPathFileSystemWatcher*

```
@Bean
        @ConditionalOnMissingBean
        ClassPathFileSystemWatcher
classPathFileSystemWatcher(FileSystemWatcherFactory
fileSystemWatcherFactory, ClassPathRestartStrategy
classPathRestartStrategy) {
      //Add watcher for files in the classpath.
            ClassPathFileSystemWatcher(fileSystemWatcherFactory,
classPathRestartStrategy, urls);
            watcher.setStopWatcherOnRestart(true);
            return watcher;
        }
```

- 2. Add listener created in step 1 to start file watching after all *beans* has been initialized.
  - **Class:** *ClassPathFileSystemWatcher.java*
  - **Function:** *afterPropertiesSet*

```
@Override
    public void afterPropertiesSet() throws Exception {
        ...
    //Add listener
        new ClassPathFileChangeListener(this.applicationContext,
this.restartStrategy, watcherToStop));

        this.fileSystemWatcher.start();
    ...
    }
```

- 3. Call *FileSystemWatcher* start a new thread to monitor and scan the changes of files.
  - **Class**: *FileSystemWatcher*.java

```
public void start() {
        synchronized (this.monitor) {
      ...
                this.watchThread.setName("File Watcher");
                this.watchThread.setDaemon(this.daemon);
                this.watchThread.start();
      ...
        }
    }
```

```
public void run() {
  ...
    scan();
  ...
}
```

```
private void scan() throws InterruptedException {
  ...
            //compare the file with current state and previous state to
find whether it changes
            while (isDifferent(previous, current));
            if (isDifferent(this.folders, current)) {
                updateSnapshots(current.values());
            }
  ...
}
```

- 4. When file changes, it will call the function `onChange` of listeners, which we registered in step 1, that is ClassPathFileChangeListener. After ClassPathFileChangeListener receives the changes, it will publish `ClassPathChangedEvent` to `LocalDevToolsAutoConfiguration` and use it to restart the server automatically.

```
public void onChange(Set<ChangedFiles> changeSet) {
        boolean restart = isRestartRequired(changeSet);
        publishEvent(new ClassPathChangedEvent(this, changeSet,
restart));
}
```

```
public void onApplicationEvent(ApplicationEvent event) {
  if (event instanceof ContextRefreshedEvent || (event instanceof
ClassPathChangedEvent
                    && !((ClassPathChangedEvent)
event).isRestartRequired())) {
            //restart!!!
                this.liveReloadServer.triggerReload();
  }
}
```