[1. Configuring Resilience4J Circuit Breakers](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/" \l "configuring-resilience4j-circuit-breakers)

[1.1. Starters](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#starters)

There are two starters for the Resilience4J implementations, one for reactive applications and one for non-reactive applications.

* org.springframework.cloud:spring-cloud-starter-circuitbreaker-resilience4j - non-reactive applications
* org.springframework.cloud:spring-cloud-starter-circuitbreaker-reactor-resilience4j - reactive applications

[1.2. Auto-Configuration](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#auto-configuration)

You can disable the Resilience4J auto-configuration by setting spring.cloud.circuitbreaker.resilience4j.enabled to false.

[1.3. Default Configuration](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#default-configuration)

To provide a default configuration for all of your circuit breakers create a Customize bean that is passed a Resilience4JCircuitBreakerFactory or ReactiveResilience4JCircuitBreakerFactory. The configureDefault method can be used to provide a default configuration.

@Bean

public Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {

return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build())

.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

.build());

}

[Reactive Example](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#reactive-example)

@Bean

public Customizer<ReactiveResilience4JCircuitBreakerFactory> defaultCustomizer() {

return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)

.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build()).build());

}

[1.4. Specific Circuit Breaker Configuration](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#specific-circuit-breaker-configuration)

Similarly to providing a default configuration, you can create a Customize bean this is passed a Resilience4JCircuitBreakerFactory or ReactiveResilience4JCircuitBreakerFactory.

@Bean

public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {

return factory -> factory.configure(builder -> builder.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2)).build()), "slow");

}

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the addCircuitBreakerCustomizer method. This can be useful for adding event handlers to Resilience4J circuit breakers.

@Bean

public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {

return factory -> factory.addCircuitBreakerCustomizer(circuitBreaker -> circuitBreaker.getEventPublisher()

.onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer), "normalflux");

}

[Reactive Example](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#reactive-example-2)

@Bean

public Customizer<ReactiveResilience4JCircuitBreakerFactory> slowCusomtizer() {

return factory -> {

factory.configure(builder -> builder

.timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2)).build())

.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()), "slow", "slowflux");

factory.addCircuitBreakerCustomizer(circuitBreaker -> circuitBreaker.getEventPublisher()

.onError(normalFluxErrorConsumer).onSuccess(normalFluxSuccessConsumer), "normalflux");

};

}

[1.5. Collecting Metrics](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#collecting-metrics)

Spring Cloud Circuit Breaker Resilience4j includes auto-configuration to setup metrics collection as long as the right dependencies are on the classpath. To enable metric collection you must include org.springframework.boot:spring-boot-starter-actuator, and io.github.resilience4j:resilience4j-micrometer. For more information on the metrics that get produced when these dependencies are present, see the [Resilience4j documentation](https://resilience4j.readme.io/docs/micrometer).

[2. Configuring Spring Retry Circuit Breakers](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#configuring-spring-retry-circuit-breakers)

Spring Retry provides declarative retry support for Spring applications. A subset of the project includes the ability to implement circuit breaker functionality. Spring Retry provides a circuit breaker implementation via a combination of it’s [CircuitBreakerRetryPolicy](https://github.com/spring-projects/spring-retry/blob/master/src/main/java/org/springframework/retry/policy/CircuitBreakerRetryPolicy.java) and a [stateful retry](https://github.com/spring-projects/spring-retry" \l "stateful-retry). All circuit breakers created using Spring Retry will be created using the CircuitBreakerRetryPolicy and a [DefaultRetryState](https://github.com/spring-projects/spring-retry/blob/master/src/main/java/org/springframework/retry/support/DefaultRetryState.java). Both of these classes can be configured using SpringRetryConfigBuilder.

[2.1. Default Configuration](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#default-configuration-2)

To provide a default configuration for all of your circuit breakers create a Customize bean that is passed a SpringRetryCircuitBreakerFactory. The configureDefault method can be used to provide a default configuration.

@Bean

public Customizer<SpringRetryCircuitBreakerFactory> defaultCustomizer() {

return factory -> factory.configureDefault(id -> new SpringRetryConfigBuilder(id)

.retryPolicy(new TimeoutRetryPolicy()).build());

}

[2.2. Specific Circuit Breaker Configuration](https://cloud.spring.io/spring-cloud-static/spring-cloud-circuitbreaker/1.0.0.RC2/reference/html/#specific-circuit-breaker-configuration-2)

Similarly to providing a default configuration, you can create a Customize bean this is passed a SpringRetryCircuitBreakerFactory.

@Bean

public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {

return factory -> factory.configure(builder -> builder.retryPolicy(new SimpleRetryPolicy(1)).build(), "slow");

}

In addition to configuring the circuit breaker that is created you can also customize the circuit breaker after it has been created but before it is returned to the caller. To do this you can use the addRetryTemplateCustomizers method. This can be useful for adding event handlers to the RetryTemplate.

@Bean

public Customizer<SpringRetryCircuitBreakerFactory> slowCustomizer() {

return factory -> factory.addRetryTemplateCustomizers(retryTemplate -> retryTemplate.registerListener(new RetryListener() {

@Override

public <T, E extends Throwable> boolean open(RetryContext context, RetryCallback<T, E> callback) {

return false;

}

@Override

public <T, E extends Throwable> void close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

}

@Override

public <T, E extends Throwable> void onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {

}

}));

}