1 Redis[1]

É um armazenamento de estruturas de dados na memória, pode ser usado como banco de dados *NoSQL*[2], *cache*[3] e outros. O Redis oferece diversas estruturas de dados como strings, hashes, listas, conjuntos e etc.

1.1 Instalação do Redis

Para instalar o Redis no Linux é muito simples, basta instalar via gerenciador de pacotes da distribuição, por exemplo no Ubuntu faz-se: *sudo apt-get install redis* (lembrar de rodar *sudo apt-get update* antes). Para instalar no Windows o caminho é ativar o WSL (Subsistema Windows para Linux) e instalar uma distribuição Linux como Ubuntu para rodar o Redis. Neste tutorial oficial da Microsoft você pode instalar o Ubuntu e realizar a instalação do Redis normalmente. Para iniciar o Redis, basta rodar o comando *redis-server*.

1.2 Como instalar o Redis Cache no Spring Boot?

É necessário em seu projeto adicionar novas dependências para utilizar o Redis corretamente, no pom.xml do seu projeto coloque as seguintes dependências:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-cache</artifactId>
   <version>2.4.3</version>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis</artifactId>
   <version>2.4.3</version>
</dependency>
```

Dessa forma seu projeto terá suporte ao cache com todas as dependências necessárias. Em seguida coloque a annotation **@EnableCaching** na classe da sua aplicação Spring.

1.3 Configurações do Redis

Crie uma classe para configurar as opções disponibilizadas pelo Redis usando a annotation **@Configuration**:

```
@Configuration
public class CacheConfig {
```

---

[1] <https://redis.io/>
[2] É um termo genérico que representa os bancos de dados não relacionais.
[3] No contexto de Redis, cache é um local reservado para armazenar informações acessadas frequentemente.

```
    @Bean
    public RedisCacheConfiguration cacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(60))
                .disableCachingNullValues()
      .serializeValuesWith(SerializationPair.fromSerializer(new GenericJackson2JsonRedisSerializer()));
    }
}
```

Dessa forma temos mais controle sobre como o Redis irá se comportar em nosso servidor usando as configurações padrões. Podemos definir o time-to-live (TTL) e alterar a estratégia de serialização do banco. Também é possível configurar opções para diferentes caches conforme sua necessidade:

```
@Bean
public RedisCacheManagerBuilderCustomizer redisCacheManagerBuilderCustomizer() {
    return (builder) -> builder
            .withCacheConfiguration("produtosCache",
                    RedisCacheConfiguration.defaultCacheConfig()
                            .entryTtl(Duration.ofMinutes(10))
                            .serializeValuesWith(SerializationPair.fromSerializer(
                                    new GenericJackson2JsonRedisSerializer())))
            .withCacheConfiguration("clientesCache",
                    RedisCacheConfiguration.defaultCacheConfig()
                            .entryTtl(Duration.ofMinutes(5))
                            .serializeValuesWith(SerializationPair.fromSerializer(
                                    new GenericJackson2JsonRedisSerializer())));
}
```

No exemplo acima foram criados dois caches customizados “produtosCache” e “clientesCache”, cada um com TTL (Time-to-Live)[4] de 10 e 5 minutos, respectivamente. Podemos usar o recurso de cache em métodos de nossos serviços que potencialmente demandam muitos custos operacionais, para fazer cache de algum método basta usar ***@Cacheable****(value = “chave”)*, no nosso caso iremos utilizar as chaves definidas acima:

```
@Component
public class ProdutoService {
    @Autowired
    private ProdutoRepository produtoRepository;

    @Cacheable(value = "produtosCache")
    public Iterable<Produto> produtos() {
        return produtoRepository.findAll();
    }
}
```

[4] Time-to-Live (TTL) é um mecanismo que limita a vida útil dos dados em um computador ou rede.

Dessa forma o cache para “produtosCache” terá duração de 10 minutos na memória, e em uma segunda vez que for chamado, ao invés de invocar o repositório será usado as informações no cache do Redis conforme a chave reservada.

1.4 Anotações importantes

Para realizar o cache com Redis, não é obrigatório configurar sempre da mesma forma como mostrado na seção anterior. A partir da configuração padrão definida em CacheConfig, podemos utilizar algumas anotações importantes:

- ***@Cacheable*** define um cache para o método, quando este é invocado o retorno dele será o que está em cache no Redis. Caso o cache tenha passado do tempo de vida, o método será invocado e a chamada será realizada de fato, salvando novamente no cache o retorno da chamada.
- ***@CachePut*** define no cache um valor com base na chave. Esta anotação não bloqueia a execução do método, é muito útil para métodos de atualização de dados.
- ***@CacheEvict*** deleta do cache um valor com base na chave. Esta anotação garante que os dados em cache estejam de acordo com as operações realizadas.

Exemplo completo:

```
@Cacheable("clientesServiceCache")
   public List<Cliente> mostrarTodosClientes() {
      return clienteRepository.findAll();
}

@Cacheable(value = "clientesServiceCache", key = "#idCliente")
   public Cliente mostrarUmCliente(Integer idCliente) {
      Optional<Cliente> cliente = clienteRepository.findById(idCliente);
      return cliente.orElseThrow();
}

@CachePut(value = "clientesServiceCache", key = "#cliente.idCliente", unless = "#result == null")
   public Cliente editarCliente(Cliente cliente) {
      mostrarUmCliente(cliente.getIdCliente());
      return clienteRepository.save(cliente);
}

@CacheEvict(value = "clientesServiceCache", key = "#idCliente")
   public void excluirCliente(Integer idCliente) {
      mostrarUmCliente(idCliente);
      clienteRepository.deleteById(idCliente);
}
```

A propriedade ***key*** é importante, pois é assim que o Redis sabe a quem mapear as informações em cache. A propriedade ***unless*** em ***@CachePut*** serve para indicar que caso o método retorne *null* o cache não será realizado. Quando se usa um valor para cache que não

está definido em ***CacheConfig***, ele irá utilizar as configurações padrões definidas pelo desenvolvedor.

1.5 Inspecionando o Redis

É possível inspecionar os valores que estão sendo armazenados em memória no Redis. Para isso, abra um terminal do WSL e digite *redis-cli* (é importante rodar redis-server antes em outra aba). Você pode usar alguns comandos para verificar as chaves armazenadas no banco:

- *KEYS **
- *GET <nome-da-chave>*