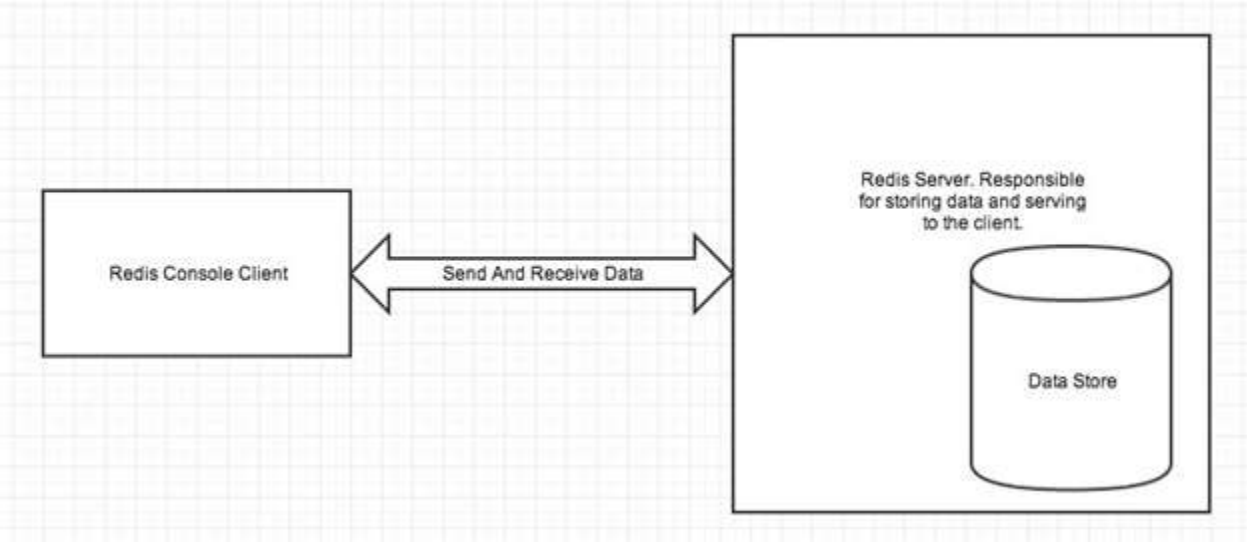
Redis

What is Redis?

- Redis is a NoSQL database which follows the principle of key-value store.
- The key-value store provides ability to store some data called a value, inside a key.
- Redis is a flexible, open-source (BSD licensed), in-memory data structure store, used as database, cache, and message broker.
- Redis is a NoSQL database so it facilitates users to store huge amount of data without the limit of a Relational database.
- Redis supports various types of data structures like strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs and geospatial indexes with radius queries.

Redis Architecture

- There are two main processes in Redis architecture:
  - Redis Client
  - Redis Server
- These client and server can be on same computer or two different computers.



Redis Architecture

- Redis server is used to store data in memory .
- It controls all type of management and forms the main part of the architecture.
- You can create a Redis client or Redis console client when you install Redis application or you can use

Features of Redis

- **Speed:** Redis stores the whole dataset in primary memory that's why it is extremely fast. It loads up to 110,000 SETs/second and 81,000 GETs/second can be retrieved in an entry level Linux box. Redis supports Pipelining of commands and facilitates you to use multiple values in a single command to speed up communication with the client libraries.
- **Persistence:** While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save. Redis supports an append-only file persistence mode.
- **Data Structures:** Redis supports various types of data structures such as strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.
- **Atomic Operations:** Redis operations working on the different Data Types are atomic, so it is safe to set or increase a key, add and remove elements from a set, increase a counter etc.

Redis vs RDBMS

- Redis stores everything in primary memory.
- RDBMS stores everything in secondary memory.

- In Redis, Read and Write operations are extremely fast because of storing data in primary memory.
- In RDBMS, Read and Write operations are slow because of storing data in secondary memory.

- Primary memory is in lesser in size and much expensive than secondary so, Redis cannot store large files or binary data.
- Secondary memory is in abundant in size and cheap than primary memory so, RDBMS can easily deal with these type of files.

Redis vs RDBMS

- Redis is used only to store those small textual information which needs to be accessed, modified and inserted at a very fast rate.
- If you try to write bulk data more than the available memory then you will receive errors.

- RDBMS can hold large data which has less frequently usage and not required to be very fast.

Redis Installation

- Go to Redis official website https://redis.io/
- https://developer.redis.com/create/windows
- Install *Ubuntu 20.04 LTS from Microsoft Store*

Redis Docker Image

- Download Redis image
  - docker pull redis
- Start Eedis docker image
  - docker run -d -p 6379:6379 redis
  - Or just use Docker desktop for start
- Check If Redis is Working
  - redis-cli
  - *redis 127.0.0.1:6379*> ping
  - PONG

Redis Set/Get

- SET some_key some_value
- GET some_key

Redis Get All Keys

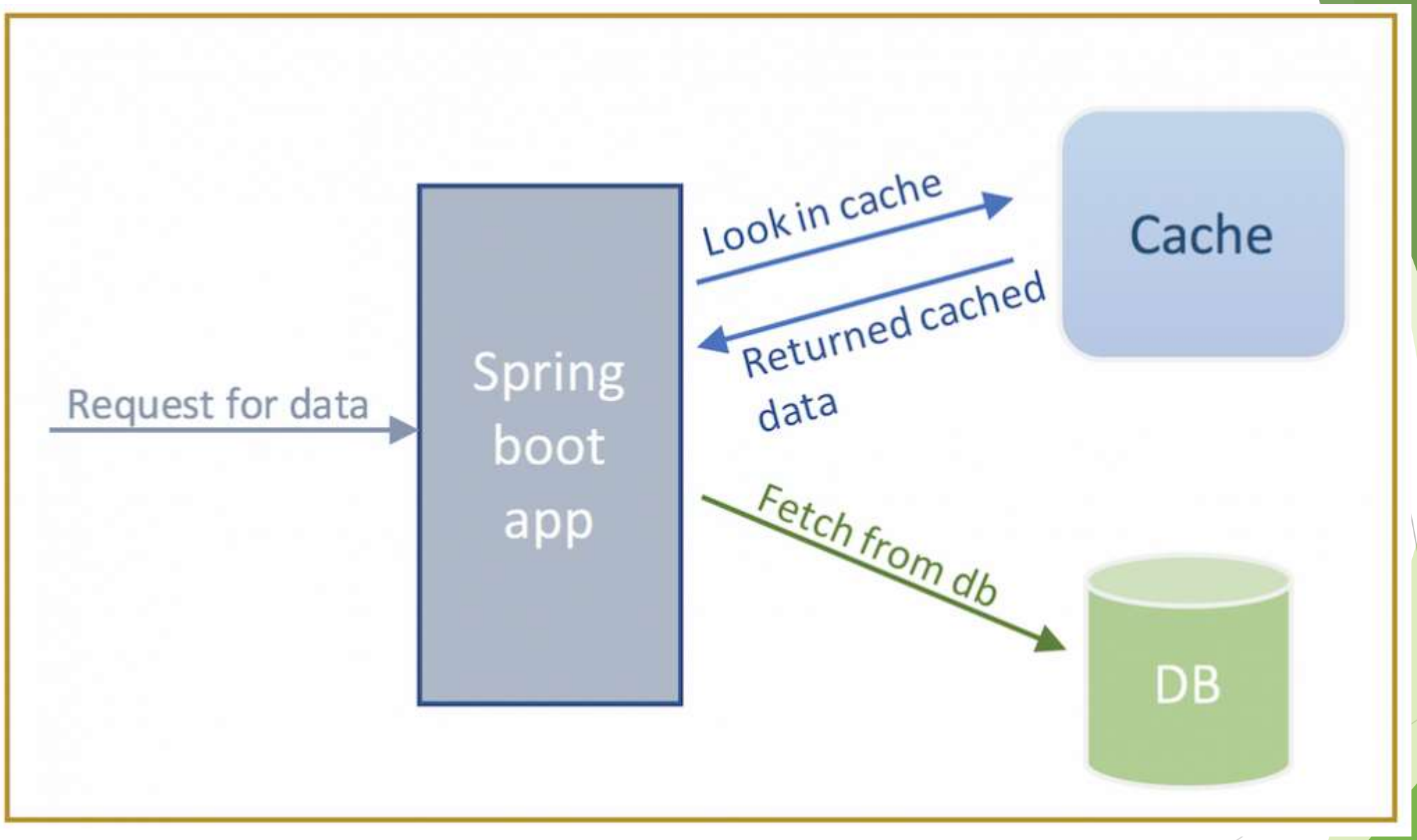
- In order to get All keys
- 1. redis-cli
- 2. keys *

Redis Get All Keys-Value

- In Redis Terminal type:

```
for i in $(redis-cli KEYS '*'); do echo $i; redis-cli GET $i; done
```

Spring Boot + Redis


Request for data
Spring boot app
Look in cache
Returned cached data
Cache
Fetch from db
DB

Spring Boot Redis

- The use of Redis Cache is are many its can be used as a database or cache but.

Redis Cache Dependancy

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

OR

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-redis</artifactId>
  <version>2.3.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.3.0</version>
  <type>jar</type>
</dependency>
```

Spring Boot Redis Config

Below is default configurations. Ularni yozmasa ham bo’ladi.

```
spring.redis.port=6379
spring.redis.password=password
spring.redis.host=localhost
```

How to use Redis Cache?

- Generally, there are four important annotations that we apply to implement Redis Cache feature in ur application. They are as below:
- **@EnableCaching**
  - We apply this annotation at the main class (starter class) of our application in order to tell Spring Container that we need Caching feature in our application.
- **@Cacheable**
  - @Cacheable is used to fetch(retrieve) data from the DB to application and store in Redis Cache. We apply it on the methods that get (retrieve) data from DB.
- **@CachePut**
  - We use @CachePut in order to update data in the Redis Cache while there is any update of data in DB. We apply it on the methods that make modifications in DB.
- **@CacheEvict**
  - We use @CachePut in order to remove data in the Redis Cache while there is any removal of data in DB. We apply it on the methods that delete data from DB.

Step 1: Add annotation @EnableCaching at starter class

```
@SpringBootApplication
@EnableCaching
public class RedisAsaCacheWithSpringBootApplication {

   public static void main(String[] args) {
      SpringApplication.run(RedisAsaCacheWithSpringBootApplication.class, args);
   }
}
```

Step 2: Create an Entity class as Invoice.java

```
@Entity(name = "profile")
public class ProfileEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column
    private String name;
    @Column
    private String surname;
}
```

@Cacheable

- This annotation is used to put value in cache for the given cache name and key @Cacheable has elements such as
- **cacheNames**: Name of the caches in which method result are stored.
- **value**: Alias for cacheNames.
- **condition**: Spring SpEL expression to make conditional caching.
- **key**: SpEL to compute key dynamically.
- **keyGenerator**: Bean name for custom KeyGenerator.
- **unless**: SpEL to veto method caching.

```
@Cacheable(value = "profileCache", key = "#id")
    public ProfileDTO get(Integer id) {
                        // get entity convert to dto
        return dto;
    }
```

@Cacheable use variation

```
@Cacheable(value="books", key="#isbn")
public Book findStoryBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="books", key="#isbn.rawNumber")
public Book findStoryBook (ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="books", key="T(classType).hash(#isbn)")
public Book findStoryBook (ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="book", condition="#name.length < 50")
public Book findStoryBook (String name)
```

@CachePut

- This annotation is used to update value in cache for the given cache name and key. @CachePut has elements such as
- **cacheNames**: Name of the caches in which method result are stored.
- **value**: Alias for cacheNames.
- **condition**: Spring SpEL expression to make conditional caching.
- **key**: SpEL to compute key dynamically.
- **keyGenerator**: Bean name for custom KeyGenerator.
- **unless**: SpEL to veto method caching.

```
@CachePut(value = "profileCache", key = "#id")
  public ProfileDTO update(Integer id, ProfileDTO dto) {
      // perform update
      return dto;
  }
```

@CacheEvict

- It is used when we need to evict (remove) the cache previously loaded of master data. When **CacheEvict** annotated methods will be executed, it will clear the cache.
- We can specify key here to remove cache, if we need to remove all the entries of the cache then we need to use allEntries=true.
- This option comes in handy when an entire cache region needs to be cleared out – rather then evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation.

@Caching

- What if we want to use multiple annotations of the same type for caching a method? Let's look at an incorrect example:

```
@Caching(evict = {
  @CacheEvict("addresses"),
  @CacheEvict(value="directory", key="#customer.name") })
public String getAddress(Customer customer) {...}
```

@CacheConfig

- With the *@CacheConfig* annotation, we can **streamline some of the cache configuration into a single place at the class level,** so that we don't have to declare things multiple times:

```
@CacheConfig(cacheNames={"addresses"})
public class CustomerDataService {
    @Cacheable
    public String getAddress(Customer customer) {...}
}
```

Customize Redis Configuration 1

- Configuration using properties file

```
spring.cache.type=redis
spring.cache.redis.cache-null-values=true
spring.cache.redis.time-to-live=4000
```

Customize Redis Configuration 2

- Configuration using java code

```
@Bean
public RedisCacheConfiguration cacheConfiguration() {
   return RedisCacheConfiguration.defaultCacheConfig()
   .entryTtl(Duration.ofMinutes(60)) // Changing the cache TTL
   .disableCachingNullValues()
   .serializeValuesWith(SerializationPair
            .fromSerializer(newGenericJackson2JsonRedisSerializer()));
}
```

Links

- https://javatechonline.com/how-to-implement-redis-cache-in-spring-boot-application/
- https://www.baeldung.com/spring-cache-tutorial
- https://www.javatpoint.com/spring-boot-caching
- https://howtodoinjava.com/spring-boot2/spring-boot-cache-example/

Spring Boot Redis As DataBase

Configure RedisTemplate

```
@Configuration
public class AppConfig {
    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new LettuceConnectionFactory();
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> empTemplate = new RedisTemplate<>();
        empTemplate.setConnectionFactory(redisConnectionFactory());
        return empTemplate;
    }
}
```

RedisTemplate

- Autowire RedisTemplate

```
@Autowired
  private RedisTemplate<String, Object> template;
```

RedisTemplate Actions

```
public Student create(Student student) {
        student.setId(UUID.randomUUID().toString());
        template.opsForHash().put(studentHash, student.getId(), student);
        return student;
    }

    public Student get(String id) {
        Student student = (Student) template.opsForHash().get(studentHash, id);
        return student;
    }

    public Student update(String id, Student dto) {
        Student student = (Student) template.opsForHash().get(studentHash, id);
        student.setName(dto.getName());
        student.setSurname(dto.getSurname());

        template.opsForHash().put(studentHash, id, dto);
        return student;
    }
```

RedisTemplate Actions

```
public List<Student> all() {
    List<Student> values = template.opsForHash().values(studentHash).stream().map(o -> {
        return (Student) o;
    }).collect(Collectors.toList());
    return values;
}


public void delete(Integer id) {
    template.opsForHash().delete(studentHash, id);
}

public void deleteAll() {
    template.delete(studentHash);
}
```

Links

- https://javatechonline.com/spring-boot-redis-crud-example/
- https://www.codeusingjava.com/boot/red
- https://medium.com/javarevisited/how-to-implement-redis-cache-using-spring-boot-c707fcf151a9