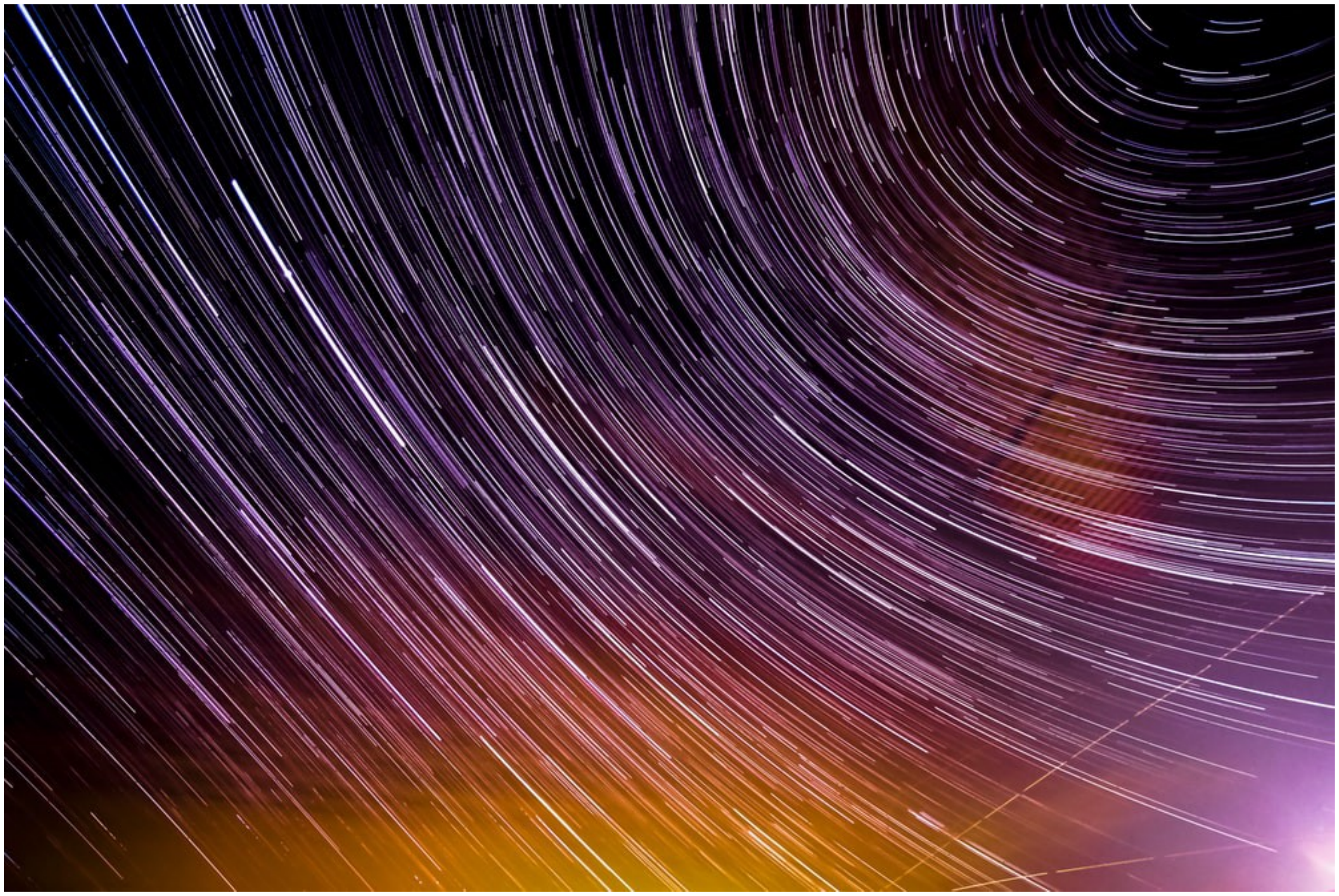
A Crash Course in Caching

An introduction to caching principles



Increasing application speed using caches

Audience

This article is aimed at developers with a reasonable understanding of how the internet functions, contemporary application architectures and AWS.

It will cover what caching is in a more general sense, the motivation behind using it, and then deep dive into specific implementations.

Argument

A cache stores previously used or computed data in a high-speed data store to make it more rapidly available than in the main storage layer. Generally we would use hardware such as RAM.

For example, we may have a preferred takeaway saved as a favourite on a delivery app. Rather than search the app for the takeaway each time, we go to our favourites section. This effectively acts as our cache.

The benefits of cache include:

- **Reduce database cost:** If your database implementation charges per IOPS (input/output per second) then a cache can help reduce this.
- **Reduce backend load:** If the same query is bombarding the backend then by introducing a cache we reduce backend load. This compounds the above point on reducing database cost as it can mean our primary data store can be smaller.
- **Improve application performance**: As mentioned previously, cache results are much quicker to retrieve.
- **Predictable performance:** High traffic events on traditional data stores can cause locking and other performance troubles. By having a high throughput cache we can reduce this.
- **Eliminate database hotspots:** Some data in our database may be more frequently accessed than others. It doesn’t make sense to scale our whole database purely based on this.

Now we have conceptually introduced cache and its benefits, let’s dive into use cases.

CDN Caching

Initially, let’s review the purpose of a content delivery network (CDN).

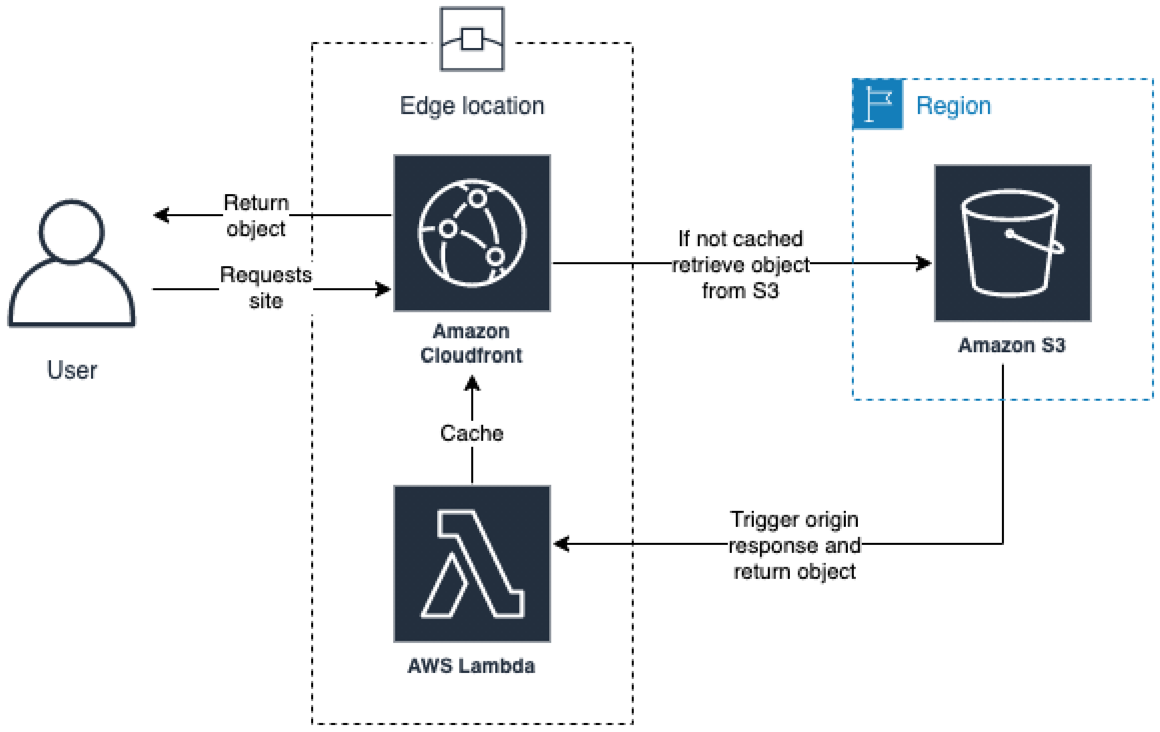
A CDN is a number of servers spread over the globe in order to provide rapid delivery of internet content. It allows for the effective loading of internet content assets such as HTML, CSS, JavaScript, videos and images.

For example, we may host our server in London, but be accessing the webpage in Philadelphia. By having a point of presence (POP) or edge server in New York, we can deliver content to Philadelphia more quickly.

Overall CDNs improve website load times, reduce bandwidth costs, increase content availability and improve security by circumventing DDOS attacks.

The AWS CDN offering is CloudFront, which can be augmented with compute power offered by Lambda@Edge. CloudFront servers acts a cache by sitting at edge locations and serving up cached versions of assets.

For example, we may be serving up a static website from an S3 bucket. We could then set up CloudFront to cache the site, and an edge Lambda to add security headers to the request.



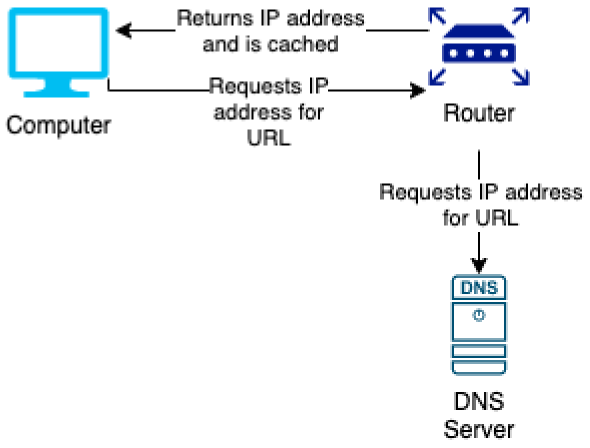
Example of using a CDN to cache information in AWS

DNS caching

Before we introduce DNS caching, let’s introduce DNS.

When we load a website in our browser we type a URL, like `https://medium.com` . Our browser then asks our router for the IP address of that website in order to retrieve the files. The router goes to a DNS server and finds the IP address, after which your browser can load the page.

A DNS cache is a temporary database on your machine which contains records of recent URL to IP address mappings.

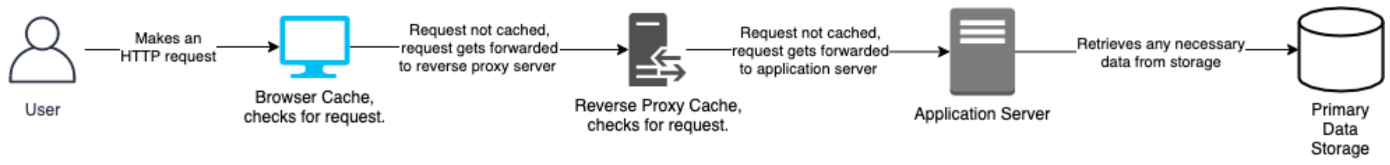

Returns IP address and is cached
Requests IP address for URL
Computer
Router
Requests IP address for URL
DNS
DNS Server

Example flow of DNS resolution and caching

It is worth noting that internally ISPs and DNS servers will do their own caching too!

Web caching

Web caching improves the responsiveness of websites by retaining HTTP responses and web resources in a cache. This cache can reside in the browser, or on the server side, which uses a reverse proxy (like Varnish) or a web application accelerator.


User
Makes an HTTP request
Browser Cache, checks for request.
Request not cached, request gets forwarded to reverse proxy server
Reverse Proxy Cache, checks for request.
Request not cached, request gets forwarded to application server
Application Server
Retrieves any necessary data from storage
Primary Data Storage

An example of the layers of HTTP Caching

Both of these (and CDNs) are driven by HTTP Caching headers. These reside on HTTP responses and allow resources to tell caches how long to store them for.

The core header to be aware of is `Cache-Control` which says who should cache the response, when, and how long for. Overall it has the form:

```
Cache-Control: public/private, max-age=0, no-cache/no-store, must-
revalidate, proxy-revalidate, no-transform
```

- **Public/ Private:** Public lets caching engines know the response can be cached, even when it is normally associated with authentication. We can indirectly imply this by supplying a max age. Private lets the browser know it can cache it, however a server cannot, as it is associated with a user's private information.

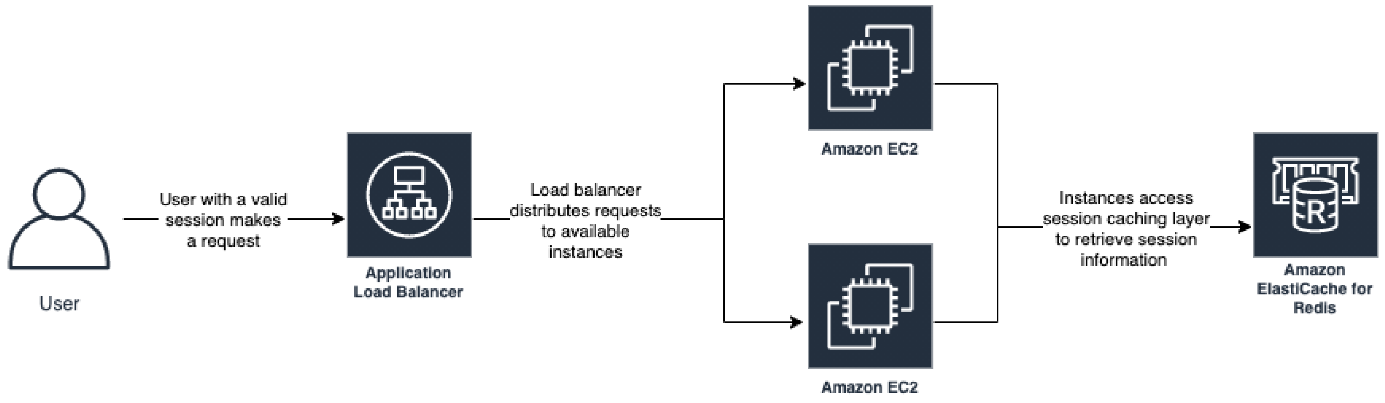
- **Max Age:** This is the maximum time in seconds a response can be used again. For example, `max-age=60` means an asset can be retrieved from cache for 60 seconds from the time the request was made.
- **No Cache/ No Store:** No cache forces a request to the server to check if it is still valid (often using a ETag header). If the cache entry is still valid it circumvents the need for a download. No store is much simpler, and prevents all caching at all levels.
- **Must revalidate:** This is used when a response becomes stale (the cache entry has run out), telling the cache it must do an end-to end validation of the response with the origin and not use stale assets.
- **Proxy revalidate:** This is the same as 'must revalidate' but only applies at the proxy level. For example, an authenticated user may have responses cached in their browser they can reuse, but any responses cached at the proxy level need to be revalidated.
- **No transform:** This tells any intermediary caches not to modify a response.

Session cache

For an overview of sessions an article is available here.

We can handle sessions in a couple of ways. One is to store a user's session on a particular node dealing with the user's requests and redirect them to this node each time (sticky sessions). Another is to have a dedicated session caching layer in an application accessed by all nodes. Often this will employ a key/ value store.

In AWS we may use something like Redis or Memcached, which we can see in the simplified AWS architecture below.


User
User with a valid session makes a request
Application Load Balancer
Load balancer distributes requests to available instances
Amazon EC2
Amazon EC2
Instances access session caching layer to retrieve session information
Amazon ElastiCache for Redis

A simplified AWS architecture showing cache used for session management.

Database cache

In memory caching can help alleviate some of the pressures on your database. If you have frequently accessed, read only data then it may make sense to store this in a cache instead of on your persistent storage. This can be done on the database itself, locally on your application servers, or like sessions it can use a dedicated caching layer.

By caching on the database we can have write-through (how updated data affects our cached values) handled automatically. However, the cache will sit on the same server as our database, meaning it will need to be scaled with the whole database engine.

By caching on our application servers we alleviate some of these issues, however we now only have access to cache on one node, which we will lose if it goes down.

A dedicated caching layer combats this, but introduces latency as requests must be made. For this and the previous solution we also face the problem of updating cache values when data is updated. A tricky logic!

General cache

You've got this gist by now, caching can be used anywhere it is useful in your application. It's not at all a silver bullet, but it can be incredibly useful!

A Quick Note on Getting Data out of Cache

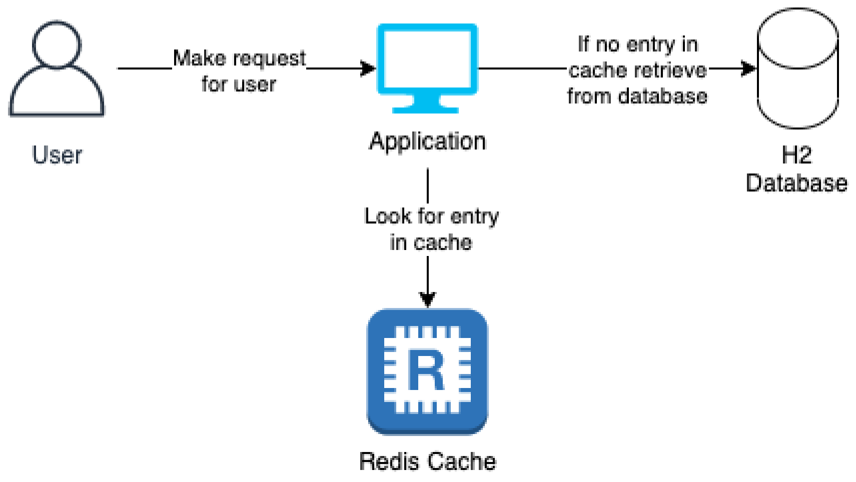
As I'm sure you've realised, cache isn't infinite. We need to get rid of entries somehow. One method is to introduce a Time to Live (TTL) — when we add an entry to our cache we give it an amount of time we would like it to hang around, after which it is deleted.

But when our cache fills up and the TTLs haven't expired yet we have further options:

- **No Eviction:** The cache throws an error.
- **FIFO (First In First Out)/ LIFO (Last In First Out):** The first/ last entry to be put in the cache is evicted.
- **LFU (Least Frequently Used)/ LRU (Least Recently Used):** The entry with the least hits/ oldest hit is evicted.
- **TLRU (Time Aware Least Recently Used):** The least recently used entry without a TTL is evicted.
- **SLRU (Segmented LRU):** The cache is split in two — probationary and protected. The first cache hit puts an entry in probationary, the second in protected. The least recently used probationary entry is evicted.
- **MRU (Most Recently Used):** The most recently used entry is evicted.
- **RR (Random Replacement):** A random entry is evicted.

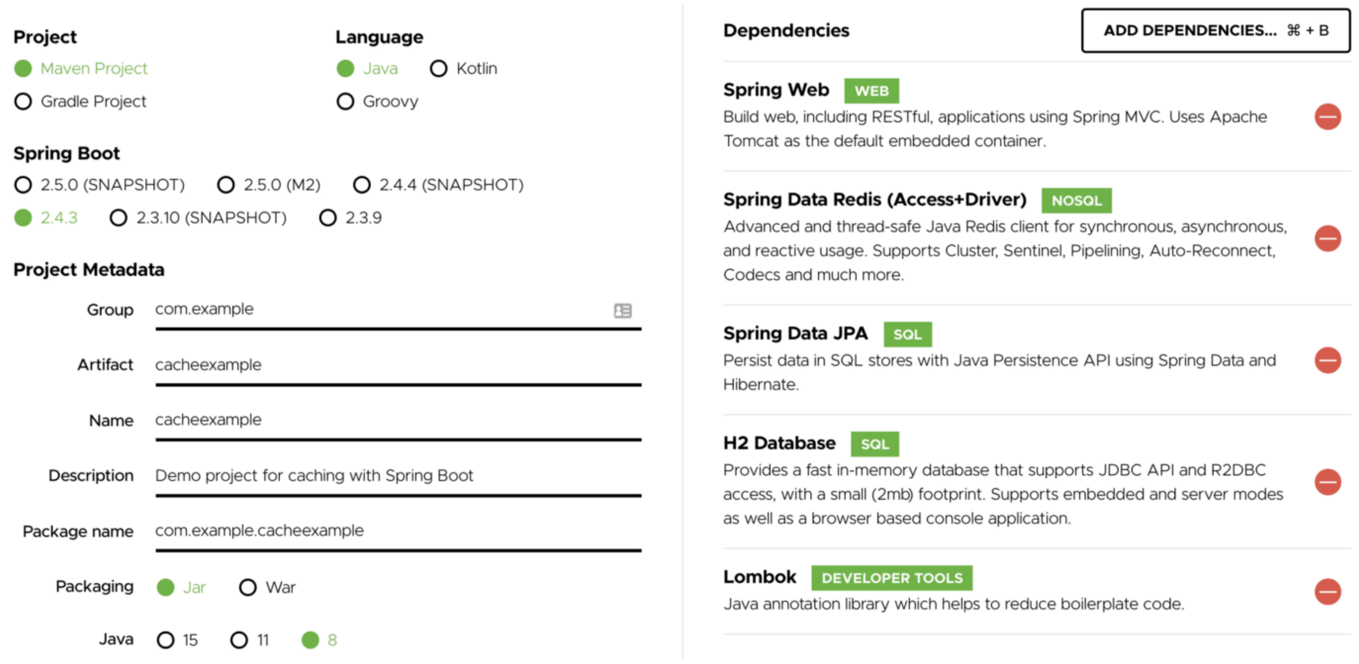
Worked Example

Let's bring this all together with an example of caching using H2, Redis, and connecting using Spring Data and Spring Data Redis. We'll implement a simple system for retrieving a user and caching them. The repository can be found here.



A basic outline of our sample application

First of all we set up the Spring project using Spring Initializr. All of the dependencies can't be shown in the one page, so check the pom file of the repository for a full list.


Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.5.0 (SNAPSHOT) ○ 2.5.0 (M2) ○ 2.4.4 (SNAPSHOT)
● 2.4.3 ○ 2.3.10 (SNAPSHOT) ○ 2.3.9

Project Metadata

Group	com.example
Artifact	cacheexample
Name	cacheexample
Description	Demo project for caching with Spring Boot
Package name	com.example.cacheexample
Packaging	● Jar ○ War
Java	○ 15 ○ 11 ● 8

Dependencies ADD DEPENDENCIES... ⌘ + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data Redis (Access+Driver) NOSQL
Advanced and thread-safe Java Redis client for synchronous, asynchronous, and reactive usage. Supports Cluster, Sentinel, Pipelining, Auto-Reconnect, Codecs and much more.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

Setting up a caching example using Spring Initializr

Once this is set up we also need to install Redis locally. I used Homebrew.

```
brew install redis
```

We then start the server running using:

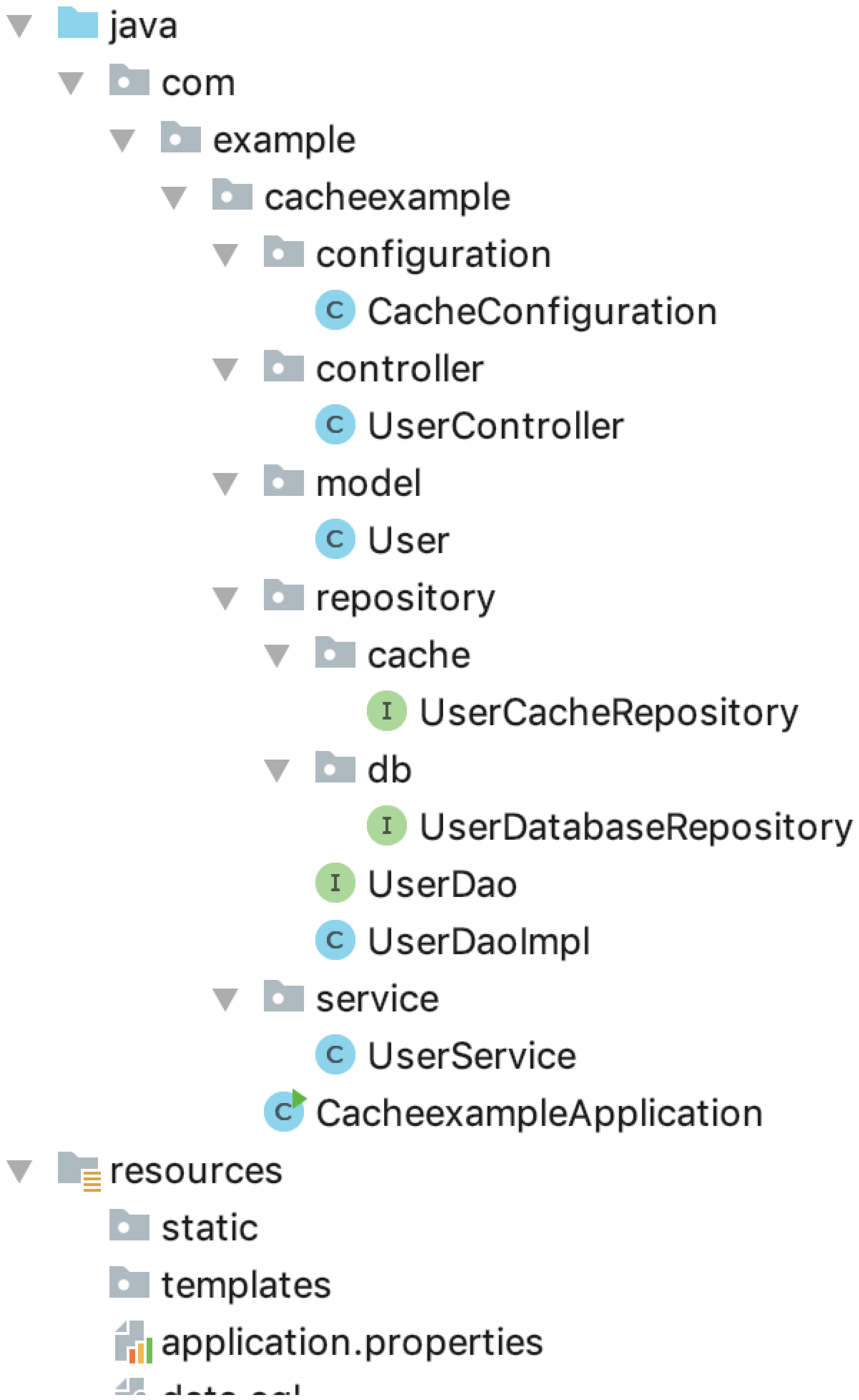
```
redis-server /usr/local/etc/redis.conf
```

We are going to implement two endpoints, one for retrieving a user which employs lazy loading, and one for persisting a new user, which uses write-through cache.

- **Lazy Loading:** This is when we put data into the cache as we load it. If I request the user with Id '1', then I will also load that into my cache.
- **Write Through:** This is when we put data into the cache as we persist it. If I create a new user with Id '2', then I will also load that into my cache.

We will also be using a TTL (time to live) for our cache entries.

The class structure can be found below:



```
java
  com
    example
      cacheexample
        configuration
          CacheConfiguration
        controller
          UserController
        model
          User
        repository
          cache
            UserCacheRepository
          db
            UserDatabaseRepository
          UserDao
          UserDaoImpl
        service
          UserService
        CacheexampleApplication
resources
  static
  templates
  application.properties
```

The Spring class structure for our example project

First of all we set up our application's main class:

```
@SpringBootApplication
@EnableJpaRepositories("com.example.cacheexample.repository.db")
@EnableRedisRepositories("com.example.cacheexample.repository.cache")
public class CacheexampleApplication {

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

}
```

We can see we're enabling repositories to connect to our database in one package, and enabling repositories to connect to Redis in another. In our `application.properties` file we turn off any automatic discovery of repositories to prevent confusion:

```
spring.data.jpa.repositories.enabled=false
spring.data.redis.repositories.enabled=false
```

Let's also look at how we're configuring our application:

```
@Configuration
public class CacheConfiguration {

    @Bean
    JedisConnectionFactory jedisConnectionFactory() {
        RedisStandaloneConfiguration redisStandaloneConfiguration = new
RedisStandaloneConfiguration("localhost", 6379);
        return new JedisConnectionFactory(redisStandaloneConfiguration);
    }

    @Bean
    public RedisTemplate<?, ?> redisTemplate() {
        RedisTemplate<?, ?> redisTemplate = new RedisTemplate<>();
         redisTemplate.setConnectionFactory(jedisConnectionFactory());
        return redisTemplate;
    }

}
```

We set Redis to connect to the local instance we just started. The `data.sql` file creates a table in our H2 database and sets up some initial data:

```
DROP TABLE IF EXISTS users;

CREATE TABLE users (
  id    INT AUTO_INCREMENT  PRIMARY KEY,
  name  VARCHAR(250) NOT NULL
);

INSERT INTO users (name) VALUES
  ('Sarah'),
  ('June'),
  ('Elizabeth');
```

Which matches our model class:

```
@Data
@Entity
@Table(name = "users")
@RedisHash(value = "User", timeToLive = 5)
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotEmpty
    private String name;

}
```

Note how we have declared a user as an entity in our database, but also as a Redis value with a 5 second TTL.

From here we expose two endpoints. One to get a user by Id (using lazy loading), one to persist a new user (with write-through).

```
@Slf4j
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/user/{id}")
    public ResponseEntity<User> getUser(@PathVariable String id) {
        log.info("Retrieving user with Id {}", id);
        return ResponseEntity.ok(userService.getUser(id));
    }

    @PostMapping("/user")
    public ResponseEntity<User> createUser(@Validated @RequestBody User
user) {
        log.info("Persisting user with name {}", user.getName());
        return ResponseEntity.ok(userService.createUser(user));
    }

}
```

Both of these pass to the service class:

```
@Slf4j
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public User getUser(String id) {
        log.info("Calling DAO to get user with Id {}", id);
        return userDao.getUser(Long.parseLong(id));
    }

    public User createUser(User user) {
        log.info("Calling DAO to persist new user with name {}",
user.getName());
        return userDao.createUser(user);
    }
}
```

Which in turn pass to the `DAO` class, which acts as a mediator between the database and the cache. This is an interface:

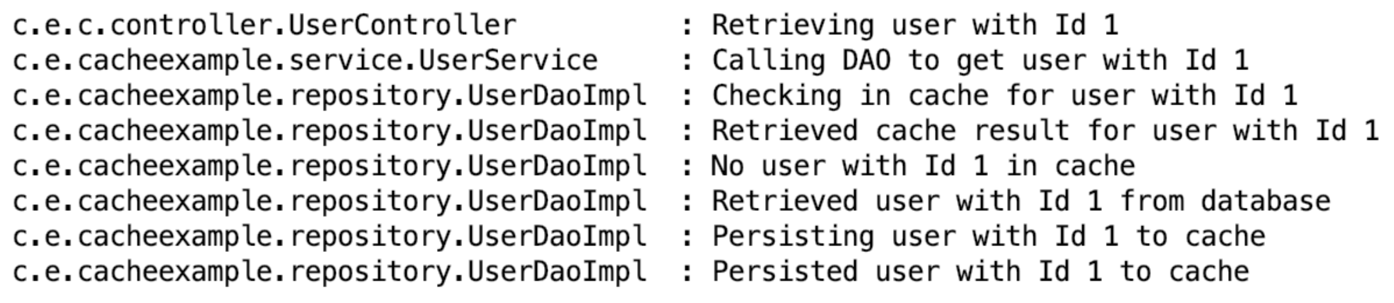
```
public interface UserDao {
    User getUser(Long id);
    User createUser(User user);
}
```

Which then has a concrete implementation.

```
@Slf4j
@Repository
public class UserDaoImpl implements UserDao {

    @Autowired
    private UserDatabaseRepository userDatabaseRepository;

    @Autowired
    private UserCacheRepository userCacheRepository;

    @Override
    public User getUser(Long id) {

        log.info("Checking in cache for user with Id {}", id);

        Optional<User> userCacheOptional =
userCacheRepository.findById(id);

        log.info("Retrieved cache result for user with Id {}", id);

        return userCacheOptional.orElseGet(() -> {

            log.info("No user with Id {} in cache", id);

            Optional<User> optionalUser =
userDatabaseRepository.findById(id);

            log.info("Retrieved user with Id {} from database", id);

            User user = optionalUser.orElseThrow(() -> new
RuntimeException("User not found"));

            log.info("Persisting user with Id {} to cache", id);
            User cacheUser = userCacheRepository.save(user);

            log.info("Persisted user with Id {} to cache",
cacheUser.getId());

            return user;

        });

    }

    @Override
    public User createUser(User user) {

        log.info("Creating a new user in the database with name {}",
user.getName());

        User newUser = userDatabaseRepository.save(user);

        log.info("Saved new user in the database Id {}, name {}",
user.getId(), user.getName());

        userCacheRepository.save(newUser);

        log.info("Saved new user in the cache Id {}, name {}",
user.getId());

        return newUser;
    }
}
```

This is what we're really interested in! In the `get` method we check to see if the user is in the cache. If not we get them from the database and put them in the cache. In the `put` method we save them to the database, then save them in the cache!

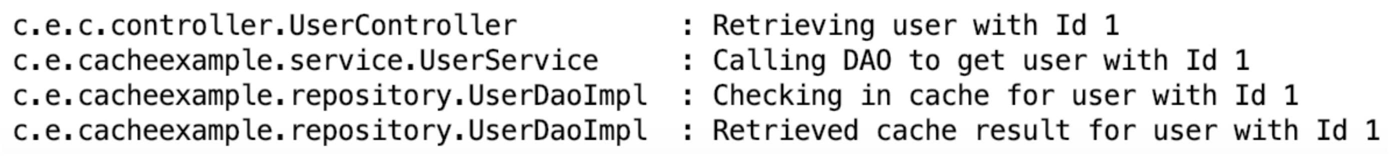
Let's have a look at it in action. First of all let's make a request to get the user with Id `1` .



```
c.e.c.controller.UserController          : Retrieving user with Id 1
c.e.cacheexample.service.UserService     : Calling DAO to get user with Id 1
c.e.cacheexample.repository.UserDaoImpl  : Checking in cache for user with Id 1
c.e.cacheexample.repository.UserDaoImpl  : Retrieved cache result for user with Id 1
c.e.cacheexample.repository.UserDaoImpl  : No user with Id 1 in cache
c.e.cacheexample.repository.UserDaoImpl  : Retrieved user with Id 1 from database
c.e.cacheexample.repository.UserDaoImpl  : Persisting user with Id 1 to cache
c.e.cacheexample.repository.UserDaoImpl  : Persisted user with Id 1 to cache
```

Log files for retrieving the user with Id 1

From the logging we can see that there was no user with Id `1` in the cache, so we have hit the database and written the result to cache. If we hit the same endpoint again in less than five seconds we receive the below:

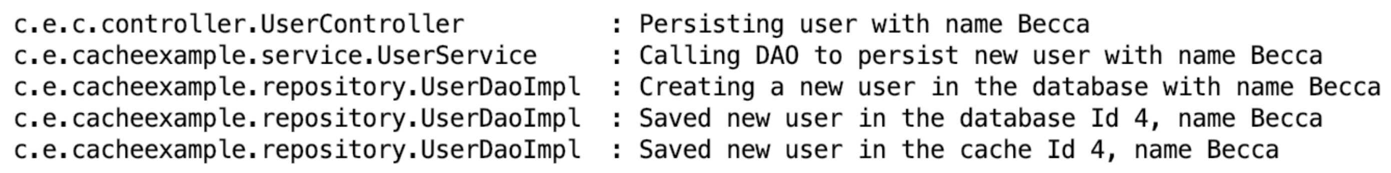


```
c.e.c.controller.UserController          : Retrieving user with Id 1
c.e.cacheexample.service.UserService     : Calling DAO to get user with Id 1
c.e.cacheexample.repository.UserDaoImpl  : Checking in cache for user with Id 1
c.e.cacheexample.repository.UserDaoImpl  : Retrieved cache result for user with Id 1
```

Log files for retrieving the cached user with Id 1

Now we no longer use the database, instead relying entirely on Redis. If we leave it for five seconds and try we're back to the first case.

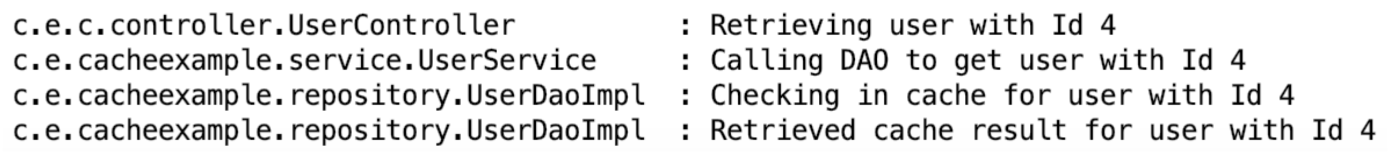
Finally, let's persist and retrieve a new user.



```
c.e.c.controller.UserController          : Persisting user with name Becca
c.e.cacheexample.service.UserService     : Calling DAO to persist new user with name Becca
c.e.cacheexample.repository.UserDaoImpl  : Creating a new user in the database with name Becca
c.e.cacheexample.repository.UserDaoImpl  : Saved new user in the database Id 4, name Becca
c.e.cacheexample.repository.UserDaoImpl  : Saved new user in the cache Id 4, name Becca
```

Log files for persisting the new user Becca

Here we can see we have persisted the user 'Becca', saving her to cache. Immediately retrieving her gives us:



```
c.e.c.controller.UserController          : Retrieving user with Id 4
c.e.cacheexample.service.UserService     : Calling DAO to get user with Id 4
c.e.cacheexample.repository.UserDaoImpl  : Checking in cache for user with Id 4
c.e.cacheexample.repository.UserDaoImpl  : Retrieved cache result for user with Id 4
```

Log files for retrieving the new user Becca from cache

Where we have retrieved her from cache.

Conclusion

In conclusion, we have covered a number of places caching can be used in a contemporary application, as well as demonstrated them in a worked example.