Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение

высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

им. Н.П. ОГАРЁВА»

(ФГБОУ ВО «МГУ им. Н.П. Огарёва»)

Факультет математики и информационных технологий

Кафедра фундаментальной информатики

КУРСОВАЯ РАБОТА

по теме:

Кеширование данных в приложениях на платформе Spring

Дисциплина Объектно-ориентированное программирование на языках высокого уровня

Автор курсовой работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М.А. Орифов

подпись, дата

Обозначение курсовой работы КР–02069964–02.03.02–14–24

Направление подготовки 02.03.02 Фундаментальная информатика и информационные технологии

Профиль Информатика и компьютерные науки

Руководитель работы

Кандидат физ.-мат. наук \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Е. Каледин

подпись, дата

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Саранск 2024

Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение

высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ

МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

им. Н.П. ОГАРЁВА»

(ФГБОУ ВО «МГУ им. Н.П. Огарёва»)

Факультет математики и информационных технологий

Кафедра фундаментальной информатики

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Орифов Мухаммаднур Аваз угли

1 Тема Кеширование данных в приложениях на платформе Spring

2 Срок предоставления работы к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3 Исходные данные для научного исследования (проектирования): учебные пособия, интернет-ресурсы

4 Содержание курсовой работы

4.1 Введение

4.2 Основные понятия

4.3 Кеширование данных в приложениях на платформе Spring

4.4 Технические детали реализации кеширования в Spring

4.5 Преимущества и недостатки кеширования данных в Spring приложениях

4.6 Сравнение с альтернативными подходами к кешированию данных

4.7 Практическая часть: разработка приложения с использованием кеширования данных в Spring

Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Е. Каледин

подпись, дата

Задание принял к исполнению \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М.А. Орифов

подпись, дата

**РЕФЕРАТ**

Курсовая работа содержит 0 страниц, 0 рисунков, 0 источников, 0 приложения.

Объектом исследования является кеширование данных и его применение в разработке приложений на платформе Spring.

Цель данной работы заключается в исследовании и анализе кеширования данных в приложениях, разработанных на платформе Spring. Работа направлена на изучение различных аспектов кеширования данных, включая его виды, интеграцию в приложения на Spring, технические детали реализации, преимущества и недостатки использования, сравнение с альтернативными подходами, а также на разработку и демонстрацию примера приложения с использованием кеша. Конечная цель состоит в формировании понимания о том, как эффективно использовать кеширование данных для повышения производительности и оптимизации работы приложений на платформе Spring.

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ ………………………………………………….………………………5

Знакомство с темой

Обоснование выбора темы

Цели и задачи исследования

1 Основные понятия…………………...………………………………………….….6

1.1 Кеширование данных: определение и принципы работы

1.2 Платформа Spring: обзор основных компонентов и возможностей

2 Кеширование данных в приложениях на платформе Spring

2.1 Виды кеширования

2.2 Интеграция кеша в приложения на Spring

2.3 Примеры использования кеширования данных в различных частях приложения

3 Технические детали реализации кеширования в Spring

3.1 Различные стратегии кеширования

3.2 Мониторинг и управление кешем

4 Преимущества и недостатки кеширования данных в Spring приложениях

4.1 Выигрыши от использования кеширования

4.2 Ограничения и проблемы при использовании кеша

6.Практическая часть: разработка приложения с использованием кеширования данных в Spring

6.1 Описание программы

6.2 Тестирование программы

ЗАКЛЮЧЕНИЕ ………………………………..…………………………………...20

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ …………..…………………..21

ПРИЛОЖЕНИЕ А (обязательное) Код разработанного приложения на Java …..22

**ВВЕДЕНИЕ**

**Знакомство с темой**

Кеширование данных является важным аспектом разработки программного обеспечения, направленным на повышение производительности и эффективности приложений. Оно представляет собой механизм временного хранения данных для быстрого доступа к ним, что позволяет уменьшить нагрузку на сервер и сократить время ответа приложения. В контексте платформы Spring, кеширование данных играет ключевую роль в оптимизации работы приложений, поскольку Spring предоставляет мощные инструменты для интеграции кеша и управления им.

**Обоснование выбора темы**

Выбор темы кеширования данных в приложениях на платформе Spring обусловлен его актуальностью и важностью для современного программирования. С увеличением объемов обрабатываемых данных и требований к производительности, эффективное использование кеширования становится необходимостью. Кроме того, Spring является одной из наиболее популярных платформ для разработки приложений на языке Java, и изучение способов интеграции кеша в эту платформу будет полезно для широкого круга разработчиков.

**Цели и задачи исследования**

Целью данной работы является изучение и анализ кеширования данных в приложениях на платформе Spring с целью определения его эффективности, преимуществ и недостатков. Для достижения этой цели ставятся следующие задачи:

1. Изучить основные концепции и принципы работы кеширования данных.

2. Рассмотреть возможности интеграции кеша в приложения на платформе Spring.

3. Провести анализ примеров использования кеширования данных в различных частях приложений на Spring.

4. Исследовать технические детали реализации кеширования в Spring, включая конфигурацию, стратегии кеширования и мониторинг.

5. Оценить преимущества и недостатки кеширования данных в приложениях на платформе Spring.

6. Провести сравнение с альтернативными подходами к кешированию данных.

7. Разработать приложение с использованием кеширования данных в Spring и провести их демонстрацию.

**1 Основные понятия**

* 1. **Кеширование данных: определение и принципы работы**

Кеширование данных представляет собой процесс временного хранения результатов выполненных операций или запросов, с целью повышения скорости доступа к этим данным. Основной идеей кеширования является сохранение часто используемых данных в быстродействующем хранилище, что позволяет избежать повторного выполнения тяжеловесных операций или запросов к источнику данных.

Принцип работы кеширования данных заключается в следующем:

* При первом запросе к данным, результат запроса сохраняется в кеше.
* При последующих запросах к этим данным, они извлекаются непосредственно из кеша, минуя выполнение дорогостоящих операций для получения данных из источника.
* Данные в кеше могут иметь ограниченное время жизни или быть удалены из кеша при изменении или обновлении данных источника.

Использование кеширования данных позволяет существенно сократить время ответа приложения и снизить нагрузку на сервер, особенно в случаях, когда данные редко изменяются или часто запрашиваются.

* 1. **Платформа Spring: обзор основных компонентов и возможностей**

Spring - это один из наиболее популярных и широко используемых фреймворков для разработки приложений на языке Java. Он предоставляет множество инструментов и компонентов, упрощающих разработку, тестирование и развертывание приложений.

Основные компоненты платформы Spring включают:

* Spring Core: основной модуль, предоставляющий базовые функциональности, такие как управление зависимостями, конфигурация и обработка исключений.
* Spring MVC: модуль для разработки веб-приложений, основанных на шаблоне MVC (Model-View-Controller).
* Spring Data: набор инструментов для работы с базами данных и другими источниками данных.
* Spring Security: модуль для обеспечения безопасности приложений, включая аутентификацию, авторизацию и защиту от атак.
* Spring Boot: инструмент для автоматизации процесса конфигурации и развертывания приложений Spring.

Платформа Spring также предоставляет возможности для интеграции кеширования данных в приложения. С помощью специальных аннотаций и конфигураций разработчики могут легко добавить кеширование в свои приложения и эффективно управлять им. Кроме того, Spring предлагает поддержку различных видов кешей и стратегий кеширования, что позволяет выбирать наиболее подходящий подход для конкретного приложения.

**2** **Кеширование данных в приложениях на платформе Spring**

**2.1 Виды кеширования**

В Spring Framework реализовано несколько основных видов кэширования для повышения производительности приложений. Вот основные виды кэширования, которые поддерживаются в Spring:

1. **In-Memory Кэширование**:

**ConcurrentMapCache**: Использует **ConcurrentHashMap** для хранения данных в памяти. Это простой вариант, подходящий для небольших приложений или для кэширования небольших объемов данных.

**Caffeine**: Более продвинутый вариант in-memory кэширования, который предлагает дополнительные функции, такие как автоматическая очистка устаревших данных и управление кэшированием.

1. **Кэширование с использованием сторонних решений**:

**EHCache**: Вариант кэширования с богатыми возможностями конфигурации и поддержки различных уровней хранения данных, включая диск и память.

**Hazelcast**: Распределённое кэширование, которое поддерживает хранение данных в кластере, что позволяет разделять кэш между несколькими узлами.

**Redis**: Кэширование с использованием Redis, который является распределённым, в памяти, с поддержкой различных структур данных.

**Memcached**: Распределённое кэширование с поддержкой простого key-value хранения в памяти.

1. **Кэширование на уровне HTTP**:

**Spring Cache Abstraction**: Это абстракция, которая поддерживает кэширование на уровне метода с использованием аннотаций, таких как **@Cacheable**, **@CachePut** и **@CacheEvict**. Это позволяет легко интегрировать кэширование с любым из вышеупомянутых решений через единый интерфейс.

**2.2 Интеграция кеша в приложения на Spring**

Интеграция кеша в приложения на платформе Spring осуществляется с помощью специальных аннотаций и конфигураций.

Для начала необходимо включить поддержку кэширования. Это можно сделать с помощью аннотации **@EnableCaching**, которую следует добавить в конфигурационный класс.

@Configuration

@EnableCaching

public class CacheConfig {

// Конфигурация кэша будет здесь

}

**Далее нужно выбрать и настроить провайдер кэша.**

Пример настройки ConcurrentMapCache:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager() {

return new ConcurrentMapCacheManager("defaultCache");

}

}

Пример настройки Redis Cache:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

Public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {

RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.defaultCacheConfig()

.serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(new StringRedisSerializer()));

return RedisCacheManager.builder(redisConnectionFactory)

.cacheDefaults(redisCacheConfiguration)

.build();

}

}

**Использование аннотаций для кэширования**

Spring предоставляет удобные аннотации для управления кэшированием.

Аннотация **@Cacheable** используется для кэширования результата метода. Когда метод аннотирован **@Cacheable** вызывается, Spring сначала проверяет, есть ли результат в кэше. Если результат есть, он возвращается из кэша, а метод не вызывается. Если результата нет, метод выполняется, а результат сохраняется в кэше.

**Основные аргументы:**

* **value** (или **cacheNames**): Имя или имена кэшей, в которых будет храниться результат метода.
* **key**: СпЕЛ-выражение (Spring Expression Language) для вычисления ключа кэша. По умолчанию используется все параметры метода.
* **condition**: СпЕЛ-выражение, которое должно возвращать **true** для кэширования. По умолчанию — всегда **true**.
* **unless**: СпЕЛ-выражение, которое предотвращает кэширование, если возвращает **true**. Проверяется после выполнения метода.
* **sync**: Если **true**, кэширование выполняется синхронно. Полезно для предотвращения конкуренции при одновременных запросах.

**Пример использования:**

@Cacheable(value = "defaultCache", key = "#id", condition = "#id > 10", unless = "#result == null")

public String getData(int id) {

// Метод выполняет дорогостоящую операцию, результат которой нужно кэшировать

return "Processed " + id;

}

Аннотация **@CachePut** используется для обновления кэша после выполнения метода. В отличие от **@Cacheable**, метод всегда выполняется, и результат его выполнения всегда помещается в кэш.

**Основные аргументы:**

* **value** (или **cacheNames**): Имя или имена кэшей, в которых будет обновлен результат метода.
* **key**: СпЕЛ-выражение для вычисления ключа кэша.
* **condition**: СпЕЛ-выражение, которое должно возвращать **true** для кэширования.
* **unless**: СпЕЛ-выражение, которое предотвращает кэширование, если возвращает **true**.

**Пример использования:**

@CachePut(value = "defaultCache", key = "#data.id", condition = "#data != null")

public String updateData(Data data) {

// Метод обновляет данные и возвращает результат, который нужно кэшировать

return "Updated " + data.getId();

}

Аннотация **@CacheEvict** используется для удаления данных из кэша. Это полезно, когда данные становятся устаревшими и их нужно удалить.

**Основные аргументы:**

* **value** (или **cacheNames**): Имя или имена кэшей, из которых будут удалены данные.
* **key**: СпЕЛ-выражение для вычисления ключа кэша, который нужно удалить.
* **allEntries**: Если **true**, все записи кэша будут удалены.
* **beforeInvocation**: Если **true**, кэш будет очищен перед выполнением метода.

**Пример использования:**

@CacheEvict(value = "defaultCache", key = "#id", allEntries = false, beforeInvocation = false)

public void deleteData(int id) {

// Метод удаляет данные, после чего нужно удалить результат из кэша

}

Аннотация **@Caching** позволяет объединять несколько кэш-аннотаций на одном методе. Это полезно, когда нужно одновременно применять несколько правил кэширования.

**Пример использования:**

@Caching(

cacheable = {

@Cacheable(value = "defaultCache", key = "#id")

},

put = {

@CachePut(value = "defaultCache", key = "#result.id")

},

evict = {

@CacheEvict(value = "anotherCache", key = "#id")

}

)

public Data getData(int id) {

// Метод логики

return new Data(id, "value");

}

Аннотация **@CacheConfig** используется на уровне класса для задания общих параметров кэширования, таких как имена кэшей и настройки ключей. Аннотации методов будут наследовать эти настройки.

**Пример использования:**

@CacheConfig(cacheNames = "defaultCache")

public class MyService {

@Cacheable(key = "#id")

public String getData(int id) {

return "Processed " + id;

}

@CachePut(key = "#result.id")

public String updateData(Data data) {

return "Updated " + data.getId();

}

@CacheEvict(key = "#id")

public void deleteData(int id) {

// Удаление данных

}

}

**2.3 Примеры использования кеширования данных в различных частях приложения**

Кеширование данных может быть использовано в различных частях приложения на платформе Spring для улучшения его производительности и эффективности. Некоторые примеры использования кеширования в Spring приложениях включают:

* Кеширование результатов SQL запросов для уменьшения нагрузки на базу данных.
* Кеширование результатов вызовов удаленных API для сокращения времени ответа.
* Кеширование часто используемых объектов или данных, таких как конфигурационные параметры или словари.
* Кеширование предварительно вычисленных или часто запрашиваемых данных для улучшения отзывчивости пользовательского интерфейса.

**3 Технические детали реализации кеширования в Spring**

**3.1 Различные стратегии кеширования**

В приложениях на платформе Spring можно использовать различные стратегии кеширования в зависимости от конкретных требований приложения и характеристик данных. Некоторые из основных стратегий кеширования в Spring включают:

**Кэширование на основе TTL (Time-To-Live)**

TTL-кэширование устанавливает срок действия кэшированных данных. После истечения заданного времени кэшированные данные считаются устаревшими и удаляются. TTL-кэширование полезно, когда данные имеют предсказуемый срок актуальности, например, кэширование курсов валют или погодных данных.

Пример использования с Redis:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public RedisCacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {

RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.defaultCacheConfig()

.entryTtl(Duration.ofMinutes(10)); // Устанавливаем TTL в 10 минут

return RedisCacheManager.builder(redisConnectionFactory)

.cacheDefaults(redisCacheConfiguration)

.build();

}

}

**Кэширование на основе LRU (Least Recently Used)**

LRU-кэширование удаляет данные, которые не использовались наиболее длительное время, когда кэш достигает предела по размеру. Это позволяет кэшу адаптироваться к изменяющимся нагрузкам и сохранять наиболее часто используемые данные. LRU-кэширование полезно в условиях ограниченной памяти, где важно поддерживать актуальность часто используемых данных.

Пример использования с Caffeine:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager() {

CaffeineCacheManager cacheManager = new CaffeineCacheManager("defaultCache");

cacheManager.setCaffeine(Caffeine.newBuilder()

.expireAfterWrite(10, TimeUnit.MINUTES)

.maximumSize(100)); // Устанавливаем максимальный размер кэша в 100 записей

return cacheManager;

}

}

**Кэширование на основе LFU (Least Frequently Used)**

LFU-кэширование удаляет наименее часто используемые данные. Это позволяет кэшу сохранять данные, которые регулярно запрашиваются. LFU-кэширование полезно в ситуациях, когда важно учитывать частоту доступа к данным, а не только их недавнее использование.

Пример использования с Caffeine:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager() {

CaffeineCacheManager cacheManager = new CaffeineCacheManager("defaultCache");

cacheManager.setCaffeine(Caffeine.newBuilder()

.expireAfterWrite(10, TimeUnit.MINUTES)

.maximumSize(100)

.evictionListener((key, value, cause) -> {

System.out.println("Evicted key: " + key + " due to " + cause);

})

.recordStats());

return cacheManager;

}

}

**Кэширование на основе Write-Through и Write-Behind**

* **Write-Through**: При записи данных в кэш, они сразу же записываются в основное хранилище. Обеспечивает согласованность данных между кэшем и основным хранилищем, полезно для критически важных данных.
* **Write-Behind**: Данные записываются в кэш и обновление основного хранилища происходит асинхронно. Улучшает производительность записи, снижая нагрузку на основное хранилище, подходит для менее критичных данных.

Пример использования с Ehcache (Write-Through):

<cache name="defaultCache"

maxEntriesLocalHeap="1000"

eternal="false"

timeToLiveSeconds="600"

memoryStoreEvictionPolicy="LRU">

<persistence strategy="localTempSwap"/>

<cacheLoaderFactory class="com.myapp.MyCacheLoaderFactory"

properties="key1=value1,key2=value2"

propertySeparator=",">

</cacheLoaderFactory>

</cache>

Пример использования с Ehcache (Write-Behind):

<cache name="defaultCache"

maxEntriesLocalHeap="1000"

eternal="false"

timeToLiveSeconds="600"

memoryStoreEvictionPolicy="LRU">

<persistence strategy="localTempSwap"/>

<cacheWriterFactory class="com.myapp.MyCacheWriterFactory"

properties="key1=value1,key2=value2"

propertySeparator=",">

</cacheWriterFactory>

<cacheWriter writeMode="write-behind"

maxWriteDelay="5"

rateLimitPerSecond="0"

writeBatching="true"

writeBatchSize="20"

retryAttempts="2"

retryAttemptDelaySeconds="5">

</cacheWriter>

</cache>

**Кэширование на уровне запросов (Query-Level Caching)**

Кэширование запросов сохраняет результаты конкретных запросов к базе данных или другому хранилищу данных. Уменьшает нагрузку на базу данных, улучшая производительность часто выполняемых запросов.

Пример использования с Hibernate:

@Entity

@Cacheable

@Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

public class MyEntity {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

Выбор стратегии кеширования зависит от различных факторов, таких как характеристики данных, требования к производительности и доступности.

**3.2 Мониторинг и управление кешем**

Мониторинг и управление кэшем являются критически важными аспектами для обеспечения эффективного использования кэш-памяти и поддержания производительности приложения. Рассмотрим различные методы мониторинга и управления кэшем, а также инструменты, которые можно использовать в Spring-приложениях:

**Встроенные инструменты мониторинга**

Spring Boot Actuator предоставляет множество эндпоинтов для мониторинга состояния приложения, включая информацию о кешах и их использовании.

Основные функции:

* Предоставление информации о текущем состоянии кэшей.
* Конфигурация кэшей в реальном времени.
* Очистка кэшей по запросу.

Endpoints для мониторинга:

/actuator/caches - список всех доступных кэшей.

/actuator/cache/{cacheName} - информация о конкретном кэше, включая размер и статистику.

**Мониторинг с использованием JMX (Java Management Extensions)**

JMX позволяет управлять и мониторить ресурсы Java-приложений, включая кэши.

Основные функции:

* Доступ к метрикам кэша через JMX консоль.
* Поддержка удаленного мониторинга.

Пример настройки JMX в Spring Boot:

В application.properties нужно написать spring.jmx.enabled=true

Конфигурируем **CacheManager** для экспорта через JMX:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager() {

return new ConcurrentMapCacheManager("defaultCache");

}

@Bean

public AnnotationMBeanExporter annotationMBeanExporter() {

return new AnnotationMBeanExporter();

}

}

**Инструменты сторонних разработчиков**

Prometheus и Grafana используются для сбора и визуализации метрик.

Основные функции:

* Сбор метрик кэша.
* Настройка алертинга.
* Визуализация данных с использованием графиков и панелей мониторинга.

Пример интеграции с Prometheus и Grafana:

1. Добавьте зависимости в **pom.xml**:

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

1. Настройте сбор метрик:

@Configuration

@EnableCaching

public class CacheConfig {

@Bean

public CacheManager cacheManager(MeterRegistry meterRegistry) {

ConcurrentMapCacheManager cacheManager = new ConcurrentMapCacheManager("defaultCache");

cacheManager.setCacheNames(List.of("defaultCache"));

return cacheManager;

}

}

1. Настройте **application.properties** для экспорта метрик:

management.endpoints.web.exposure.include=prometheus management.endpoint.prometheus.enabled=true

Эффективный мониторинг и управление кэшем играют ключевую роль в обеспечении высокой производительности и надежности приложений. Инструменты, такие как Spring Boot Actuator, JMX, Prometheus и Grafana, предоставляют разработчикам возможность получать подробную информацию о состоянии кэшей и управлять ими. Программное управление и аннотации Spring обеспечивают дополнительные возможности для гибкого контроля над кэшированием. Эти методы позволяют разработчикам эффективно отслеживать использование кэша, выявлять проблемы производительности и оперативно принимать меры для их решения.

**4 Преимущества и недостатки кеширования данных в Spring приложениях**

**4.1 Выигрыши от использования кеширования**

Кеширование данных в приложениях на платформе Spring может принести ряд значительных выгод:

**1. Улучшение производительности:** Кэширование позволяет значительно сократить время отклика приложений за счет уменьшения количества запросов к базе данных или другим источникам данных. Например, часто запрашиваемые данные могут быть извлечены из кэша, что обеспечивает мгновенный доступ по сравнению с выполнением сложных запросов к базе данных.

**2. Снижение нагрузки на сервер:** За счет уменьшения количества обращений к базе данных и другим ресурсам, кэширование снижает нагрузку на сервер, что позволяет улучшить масштабируемость системы. В условиях высокой нагрузки это помогает избежать перегрузки серверов и падений производительности.

**3. Повышение доступности данных:** Кэширование позволяет обеспечить доступ к данным даже в случае временной недоступности основного хранилища. Это особенно важно для приложений, требующих высокой доступности и быстрого отклика, таких как интернет-магазины или новостные сайты.

**4. Оптимизация использования ресурсов:** Эффективное кэширование позволяет более рационально использовать вычислительные ресурсы, такие как CPU и память. Вместо выполнения тяжелых вычислительных операций каждый раз, приложение может обращаться к кэшу для получения уже рассчитанных данных.

**4.2 Ограничения и проблемы при использовании кеша**

Однако кеширование данных в Spring приложениях может также иметь свои ограничения и проблемы:

**1. Устаревшие данные:** Одна из главных проблем кэширования - это возможность возврата устаревших данных, особенно в системах с высокой частотой обновления данных. Это может привести к несоответствиям и ошибкам в приложении.

**2. Затраты на управление кэшем:** Управление кэшом требует дополнительных ресурсов и внимания. Настройка, мониторинг и обслуживание кэша могут быть сложными задачами, требующими понимания внутренней логики работы кэша и его оптимизации.

**3. Проблемы синхронизации:** В распределенных системах поддержание согласованности данных между кэшами может быть сложной задачей. Проблемы могут возникнуть, когда несколько экземпляров приложения работают с различными версиями данных, что может привести к рассогласованности данных.

**4. Ограничения памяти:** Кэширование требует выделения дополнительной памяти для хранения данных. В системах с ограниченными ресурсами это может стать проблемой, так как большое количество кэшированных данных может привести к исчерпанию доступной памяти и снижению общей производительности системы.

Кэширование в Spring-приложениях предоставляет значительные преимущества, такие как улучшение производительности, снижение нагрузки на сервер, повышение доступности данных и оптимизация использования ресурсов. Однако, наряду с этими преимуществами, существуют ограничения и проблемы, такие как устаревшие данные, затраты на управление кэшем, проблемы синхронизации и ограничения памяти. Для эффективного использования кэширования необходимо тщательно продумывать стратегию кэширования, выбирать подходящие инструменты и технологии, а также регулярно мониторить и оптимизировать работу кэша.

**6 Практическая часть: разработка приложения с использованием кеширования данных в Spring**

**6.1 Описание программы**

В этой части работы мы создадим простое Spring Boot приложение с использованием кеширования данных. Это приложение будет представлять собой REST API для управления книгами в библиотеке, где данные о книгах будут кешироваться для повышения производительности. Наше приложение будет выполнять следующие основные функции:

* Добавление новой книги.
* Получение информации о книге по ее идентификатору.
* Получение списка всех книг.

Мы будем использовать кеширование для хранения информации о книгах, чтобы ускорить операции чтения. На рисунке 1 изображен Spring Initializr (https://start.spring.io/) для создания нового проекта с зависимостями Spring Web, Spring Data JPA, H2 Database и Spring Cache

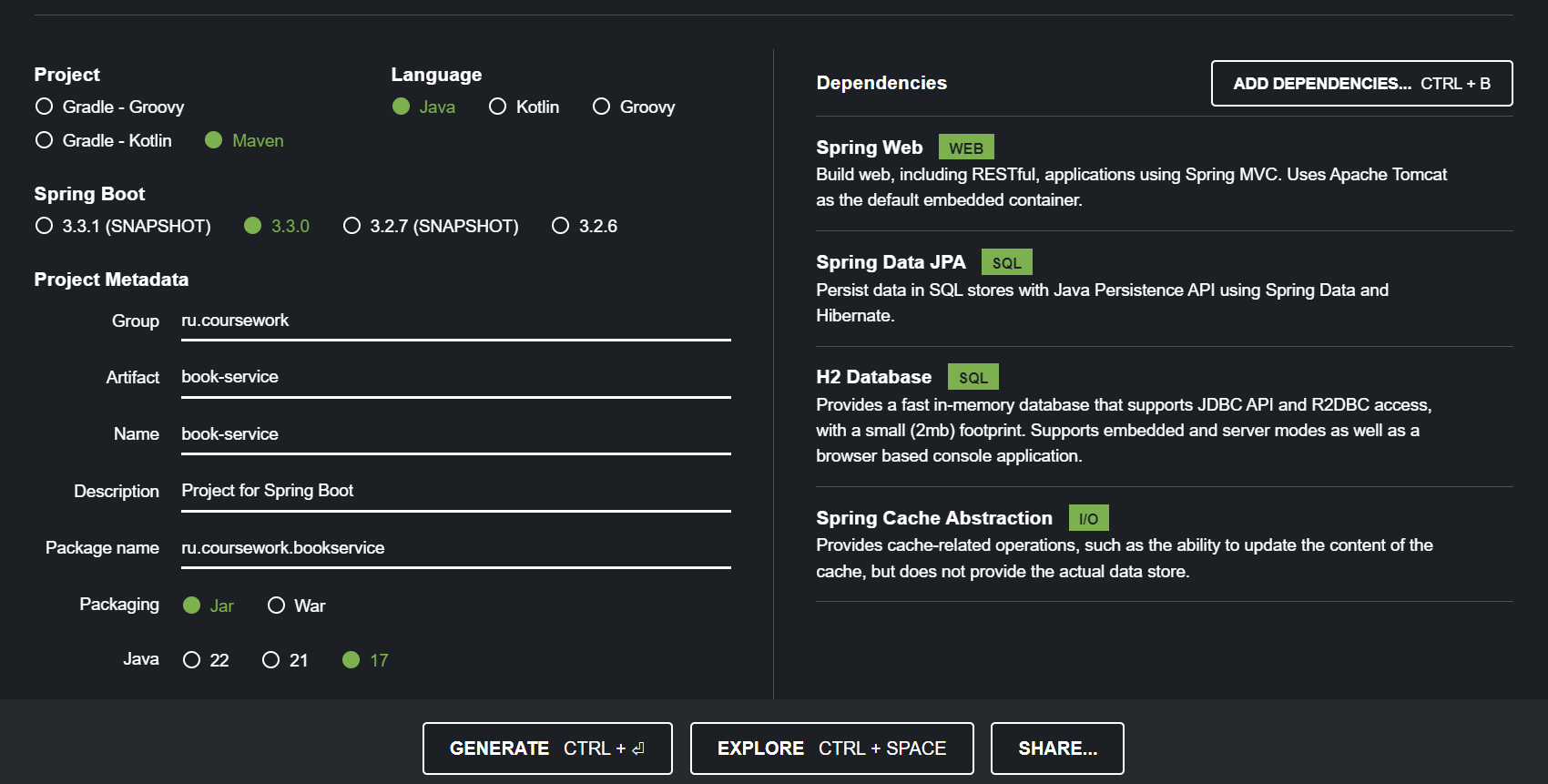


Рисунок 1 – создания нового проекта с нужными зависимостями

**Включаем кеширование в основном классе приложения:**

@SpringBootApplication

@EnableCaching

public class BookServiceApplication {

public static void main(String[] args) {

SpringApplication.run(BookServiceApplication.class, args);

}

}

**Создание сущности Book:**

@Data

@Entity

@AllArgsConstructor

@NoArgsConstructor

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

}

**Создание репозитория BookRepository:**

@Repository

public interface BookRepository extends JpaRepository<Book, Long> {

}

**Создание сервиса BookService:**

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

@Cacheable("books")

public Book getBookById(Long id) {

return bookRepository.findById(id).orElse(null);

}

@CacheEvict(value = "books", allEntries = true)

public Book addBook(Book book) {

return bookRepository.save(book);

}

@Cacheable("books")

public List<Book> getAllBooks() {

return bookRepository.findAll();

}

}

**Создание контроллера BookController:**

@RestController

@RequestMapping("/books")

public class BookController {

@Autowired

private BookService bookService;

@PostMapping

public Book addBook(@RequestBody Book book) {

return bookService.addBook(book);

}

@GetMapping("/{id}")

public Book getBookById(@PathVariable Long id) {

return bookService.getBookById(id);

}

@GetMapping

public List<Book> getAllBooks() {

return bookService.getAllBooks();

}

}

**6.2 Тестирование программы**

Для тестирования нашего приложения мы создадим несколько тестов с использованием JUnit и Spring Boot Test. Мы проверим работу кеширования и корректность выполнения основных операций.

**Тестирование BookService:**

@RunWith(SpringRunner.class)

@SpringBootTest

public class BookServiceTest {

@Autowired

private BookService bookService;

@Autowired

private BookRepository bookRepository;

@Test

public void testAddAndGetBook() {

Book book = new Book();

book.setTitle("Test Book");

book.setAuthor("Test Author");

bookService.addBook(book);

Book fetchedBook = bookService.getBookById(book.getId());

assertEquals(book.getTitle(), fetchedBook.getTitle());

assertEquals(book.getAuthor(), fetchedBook.getAuthor());

}

@Test

public void testGetAllBooks() {

bookService.addBook(new Book(1L, "Book 1", "Author 1"));

bookService.addBook(new Book(2L,"Book 2", "Author 2"));

List<Book> books = bookService.getAllBooks();

assertEquals(2, books.size());

}

@Test

public void testCacheableGetBookById() {

Book book = new Book();

book.setTitle("Cached Book");

book.setAuthor("Cached Author");

bookService.addBook(book);

Book firstFetch = bookService.getBookById(book.getId());

Book secondFetch = bookService.getBookById(book.getId());

verify(bookRepository, times(1)).findById(book.getId());

}

}

**Тестирование контроллера BookController:**

@RunWith(SpringRunner.class)

@SpringBootTest

@FixMethodOrder(MethodSorters.NAME\_ASCENDING)

@AutoConfigureMockMvc

public class BookControllerTest {

@Autowired

private MockMvc mockMvc;

@Autowired

private BookRepository bookRepository;

@Test

public void test3AddBook() throws Exception {

Book book = new Book();

book.setTitle("New Book");

book.setAuthor("New Author");

mockMvc.perform(post("/books")

.contentType(MediaType.APPLICATION\_JSON)

.content(new ObjectMapper().writeValueAsString(book)))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("New Book"))

.andExpect(jsonPath("$.author").value("New Author"));

}

@Test

public void test2GetBookById() throws Exception {

Book book = new Book();

book.setTitle("Another Book");

book.setAuthor("Another Author");

book = bookRepository.save(book);

mockMvc.perform(get("/books/" + book.getId()))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Another Book"))

.andExpect(jsonPath("$.author").value("Another Author"));

}

@Test

public void test1GetAllBooks() throws Exception {

Book entity1 = new Book();

Book entity2 = new Book();

entity1.setAuthor("Author 1");

entity1.setTitle("Book 1");

entity2.setAuthor("Author 2");

entity2.setTitle("Book 2");

bookRepository.save(entity1);

bookRepository.save(entity2);

mockMvc.perform(get("/books"))

.andExpect(status().isOk())

.andExpect(jsonPath("$", hasSize(2)));

}

}

В данной практической части мы разработали простое Spring Boot приложение для управления книгами с использованием кеширования данных. Мы рассмотрели основные этапы разработки, включая создание проекта, конфигурацию кеша, разработку сущностей, репозиториев, сервисов и контроллеров. Мы также провели тестирование программы, чтобы убедиться в корректности работы кеширования и выполнения основных операций.

**ЗАКЛЮЧЕНИЕ**

**Подведение итогов исследования**

В ходе данного исследования были рассмотрены основные аспекты кеширования данных в приложениях, разработанных на платформе Spring. Мы углубились в понятие кеширования данных и его значимость для повышения производительности и эффективности приложений. Кроме того, было осуществлено ознакомление с основными компонентами платформы Spring, что позволило лучше понять способы интеграции кеша и его управления в приложениях на данной платформе.

**Выводы по результатам работы**

Исходя из поставленных целей и задач исследования, мы приходим к следующим выводам:

Кеширование данных в приложениях на платформе Spring является важным аспектом, способствующим повышению производительности и эффективности систем.

Интеграция кеша в приложения на Spring предоставляет разработчикам мощные инструменты для управления данными и оптимизации их обработки.

Анализ различных аспектов кеширования данных, включая его технические детали реализации, преимущества и недостатки, позволяет получить глубокое понимание механизмов работы кеша в Spring-приложениях.

Практическая часть работы, включающая разработку примера приложения с использованием кеширования данных в Spring, демонстрирует применение полученных знаний на практике и помогает лучше понять принципы работы кеша в реальных условиях.

**Рекомендации по использованию кеширования данных в приложениях на платформе Spring**

По результатам исследования рекомендуется следующее:

Разработчикам приложений на платформе Spring следует активно использовать возможности кеширования данных для повышения производительности и эффективности своих систем.

При выборе стратегий кеширования и настройке параметров кеша необходимо учитывать особенности конкретного приложения и его требования к производительности.

Постоянное мониторинг и анализ производительности кеша поможет оптимизировать его работу и достичь максимальной эффективности.

При разработке приложений следует также учитывать возможность адаптации кеша к изменяющимся условиям и требованиям системы.

В целом, кеширование данных в приложениях на платформе Spring представляет собой мощный инструмент для оптимизации работы систем и повышения их производительности. Соблюдение рекомендаций по использованию кеша позволит разработчикам создавать более эффективные и отзывчивые приложения, способные успешно справляться с растущими объемами данных и повышенными требованиями к производительности.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1 Johnson, R. (2018). Spring in Action (5th ed.). Manning Publications.

2. Wall, C., Fisher, R., & Loughran, S. (2005). Spring Framework Documentation. [Online] Available at: <https://docs.spring.io/spring-framework/docs/current/reference/html/>.

3. White, J. (2019). Spring Boot in Action. Manning Publications.

4. Gupta, A. (2020). Mastering Spring 5.0. Packt Publishing.

5. JavaDocs for Spring Framework. [Online] Available at: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/>.

6. Official documentation for Ehcache. [Online] Available at: <https://www.ehcache.org/documentation/>.

7. Official documentation for Caffeine Cache. [Online] Available at: <https://github.com/ben-manes/caffeine/wiki>.

8. Schabell, E. (2018). Hands-On Spring Security 5 for Reactive Applications. Packt Publishing.

9. Введение в кеширование данных в Spring. [Online] Available at: <https://www.baeldung.com/spring-cache-tutorial>.

10. Официальная документация Spring Boot. [Online] Available at: <https://docs.spring.io/spring-boot/docs/current/reference/html/>.

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Код разработанного приложения на Java**

**ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Код подключения зависимостей для Кэш**