



Lesson 29

01.09.2022

```
public class Test1 {  
    public static void main(String[] args) {  
        Test1 test = new Test1();  
        System.out.println(test == this);  
    }  
}
```

```
public class Test2 {  
    public static void main(String[] args) {  
        Test2 test = new Test2();  
        test.print("C");  
    }  
    public void print() {  
        System.out.println("A");  
    }  
    public static void print(String s) {  
        System.out.println("B");  
    }  
}
```



```
public class Test3 {  
    public static void main(String[] args) {  
        Set<Number> set = new HashSet<>();  
        set.add(1);  
        set.add(1L);  
        set.add(1.0);  
        System.out.println(set.size());  
    }  
}
```

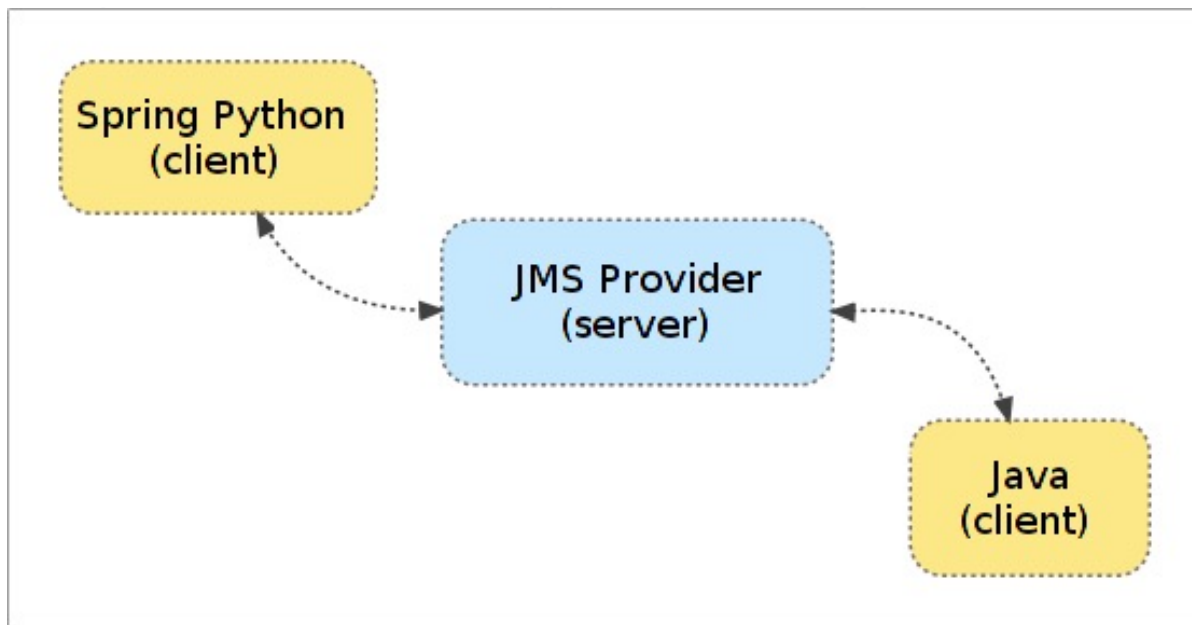
```
class MyLink{
    public MyLink(){
        str = "New";
    }
    public String str;
}


public class Test4{
    public static void main(String[] args) {
        MyLink b1 = new MyLink();
        MyLink b2 = b1;
        b2.str = "MyString";
        System.out.println(b1.str);

        String a1 = "Test";
        String a2 = a1;
        System.out.println(a2);
        a1 = "Not a Test";
        System.out.println(a2);
    }
}
```

Java Message Service

Java Message Service (JMS) — стандарт промежуточного ПО для рассылки сообщений, позволяющий приложениям создавать, посылать, получать и читать сообщения.





Коммуникация между компонентами, использующими JMS, *асинхронна* (процедура не дожидается ответа на своё сообщение) и независима от исполнения компонентов.

JMS поддерживает две модели обмена сообщениями: «точка - точка» и «издатель-подписчик».

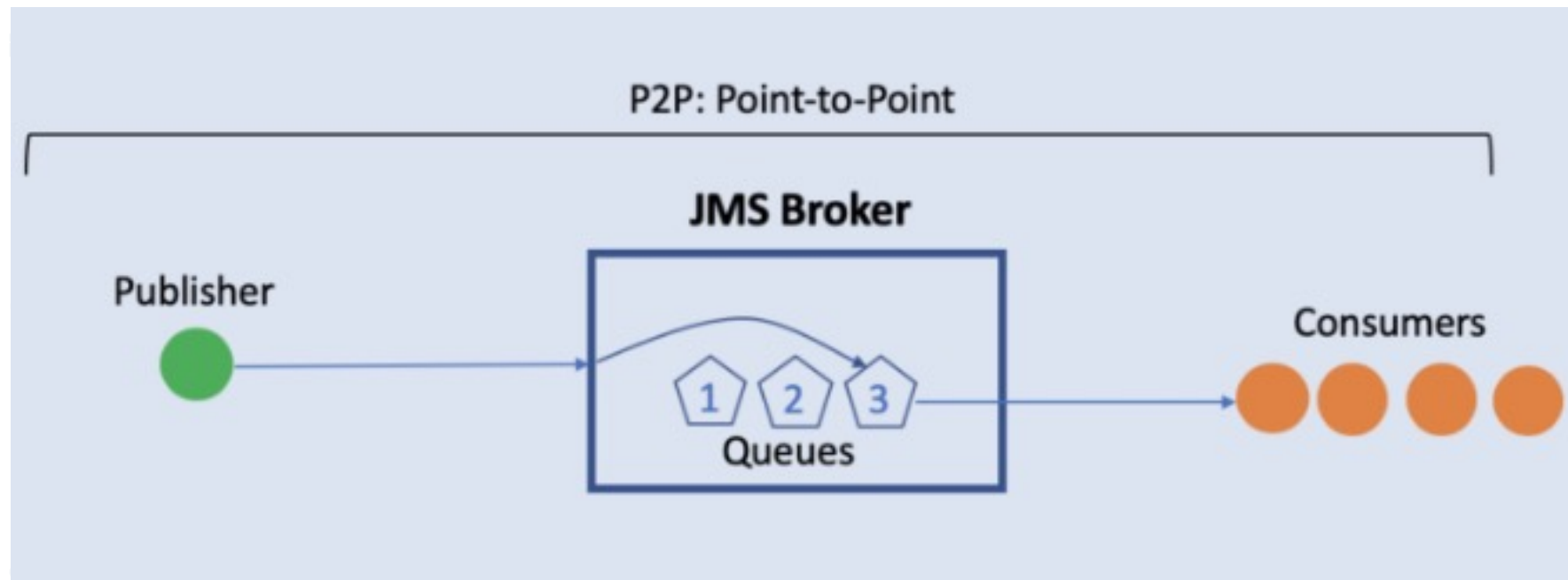
Модель **«точка - точка»** характеризуется следующим:

- Каждое сообщение имеет только одного адресата
- Сообщение попадает в «очередь» адресата и может быть прочитано когда угодно. Если адресат не работал в момент отсылки сообщения, сообщение не пропадёт.
- После получения сообщения адресат посылает извещение.

Модель **«издатель-подписчик»** характеризуется следующим:

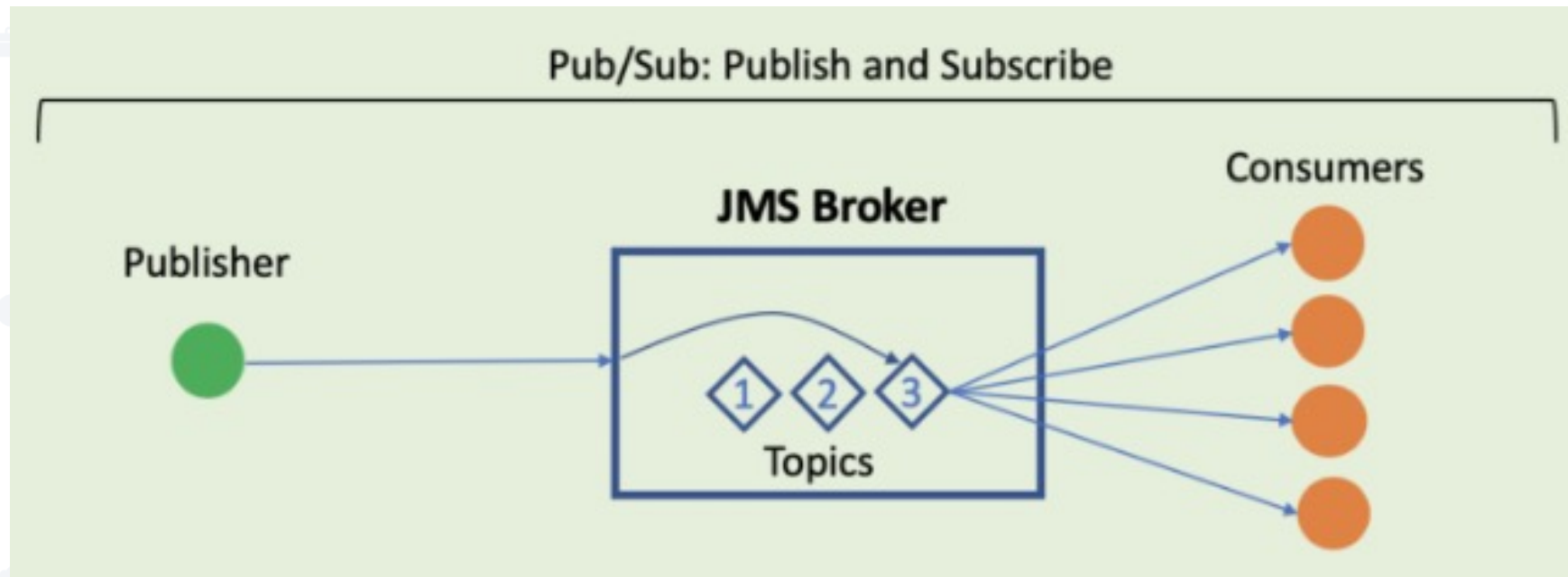
- Подписчик подписывается на определённую «тему»
- Издатель публикует своё сообщение. Его получают все подписчики этой темы
- Получатель должен работать и быть подписан в момент отправки сообщения

«Точка - точка»





«издатель-подписчик»





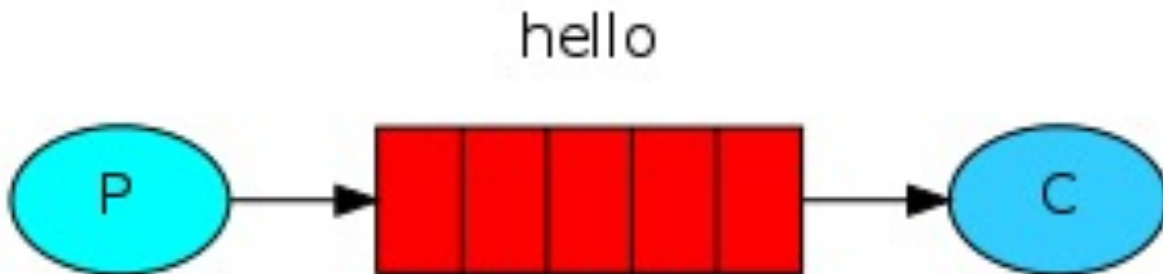
 RabbitMQ

 ActiveMQ



RabbitMQ

RabbitMQ – это брокер сообщений. Его основная цель – принимать и отдавать сообщения. Его можно представлять себе, как почтовое отделение: когда Вы бросаете письмо в ящик, Вы можете быть уверены, что рано или поздно почтальон доставит его адресату. В этой аналогии RabbitMQ является одновременно и почтовым ящиком, и почтовым отделением, и почтальоном.





Терминология

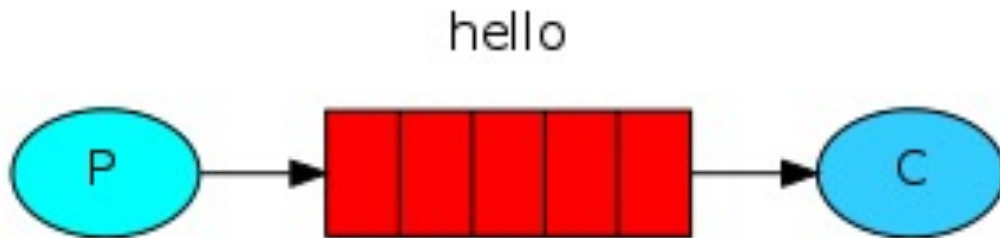
Producer (поставщик) – программа, отправляющая сообщения. В схемах он будет представлен кругом с буквой «Р».

Queue (очередь) – имя «почтового ящика». Она существует внутри RabbitMQ. Хотя сообщения проходят через RabbitMQ и приложения, хранятся они только в очередях. Очередь не имеет ограничений на количество сообщений, она может принять сколь угодно большое их количество – можно считать ее бесконечным буфером. Любое количество поставщиков может отправлять сообщения в одну очередь, также любое количество подписчиков может получать сообщения из одной очереди. В схемах очередь будет обозначена стеком и подписана именем (HELLO).

Consumer (подписчик) – программа, принимающая сообщения. Обычно подписчик находится в состоянии ожидания сообщений. В схемах он будет представлен кругом с буквой «С»:

Hello World!

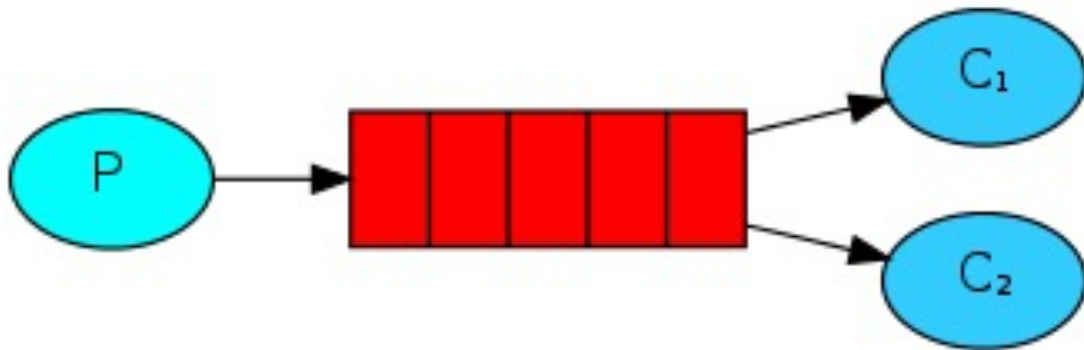
Первый пример не будет особо сложным – давайте просто отправим сообщение, примем его и выведем на экран. Для этого нам потребуется две программы: одна будет отправлять сообщения, другая – принимать и выводить их на экран. Общая схема такова:



<https://www.rabbitmq.com/download.html>

Work Queues

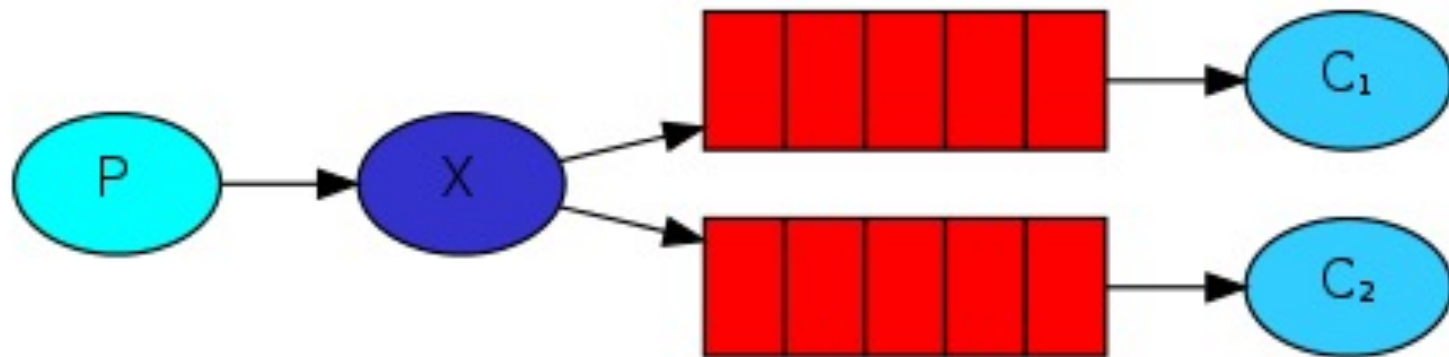
В данном примере одну очередь слушают уже два листенера. Для эмуляции полезной работы используем `Thread.sleep`. Важно, что листенеры одной очереди могут быть и на разных экземплярах программы. Так можно распараллелить очередь на несколько компьютеров или нод в облаке.



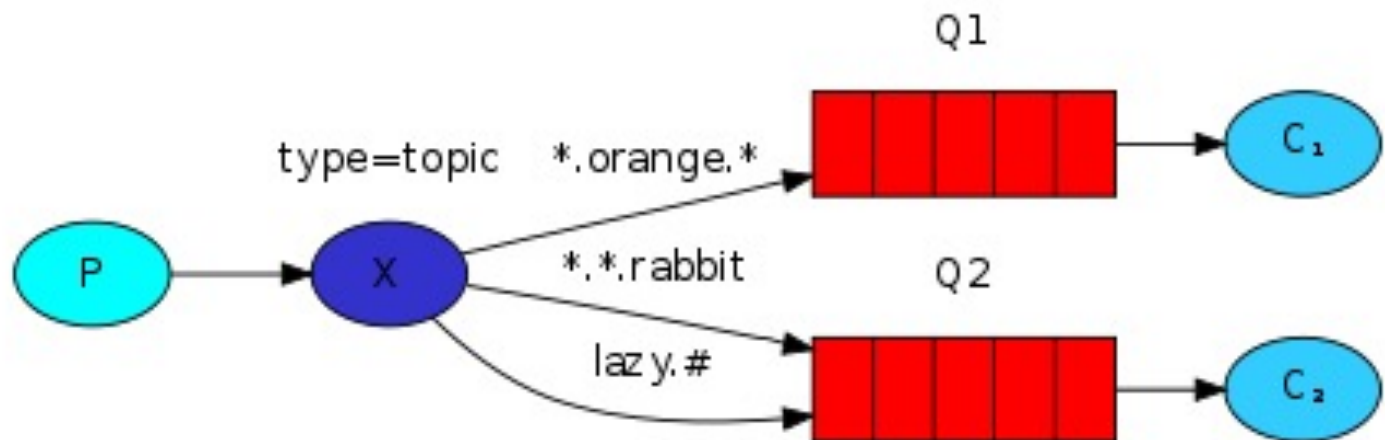


Publish/Subscribe

Тут одно и то же сообщение приходит сразу двум консьюмерам.

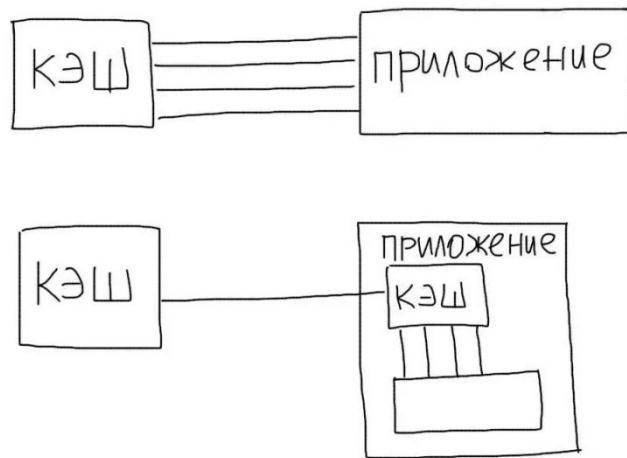


Topics



Что такое кэширование?

В сфере вычислительной обработки данных кэш – это высокоскоростной уровень хранения, на котором требуемый набор данных, как правило, временного характера. Доступ к данным на этом уровне осуществляется значительно быстрее, чем к основному месту их хранения. С помощью кэширования становится возможным эффективное повторное использование ранее полученных или вычисленных данных.





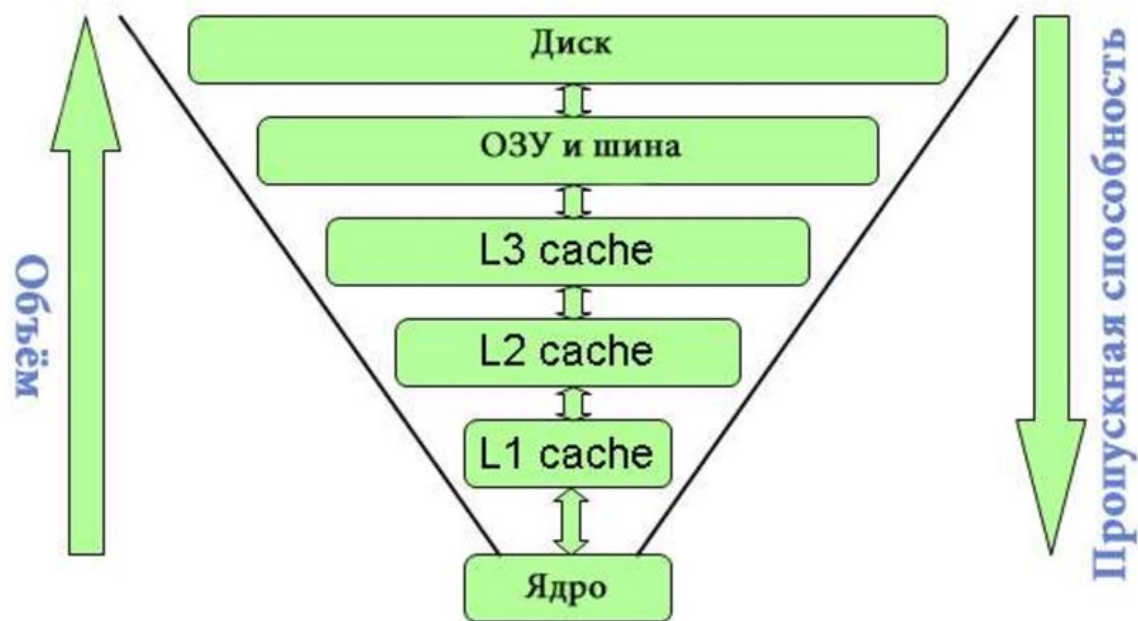
Как работает кэширование?

Данные в кэше обычно хранятся на устройстве с быстрым доступом, таком как ОЗУ (оперативное запоминающее устройство), и могут использоваться совместно с программными компонентами. Основная функция кэша – ускорение процесса извлечения данных. Он избавляет от необходимости обращаться к менее скоростному базовому уровню хранения.

Небольшой объем памяти кэша компенсируется высокой скоростью доступа. В кэше обычно хранится только требуемый набор данных, причем временно, в отличие от баз данных, где данные обычно хранятся полностью и постоянно.

Уровни кэша

Уровни кэш-памяти





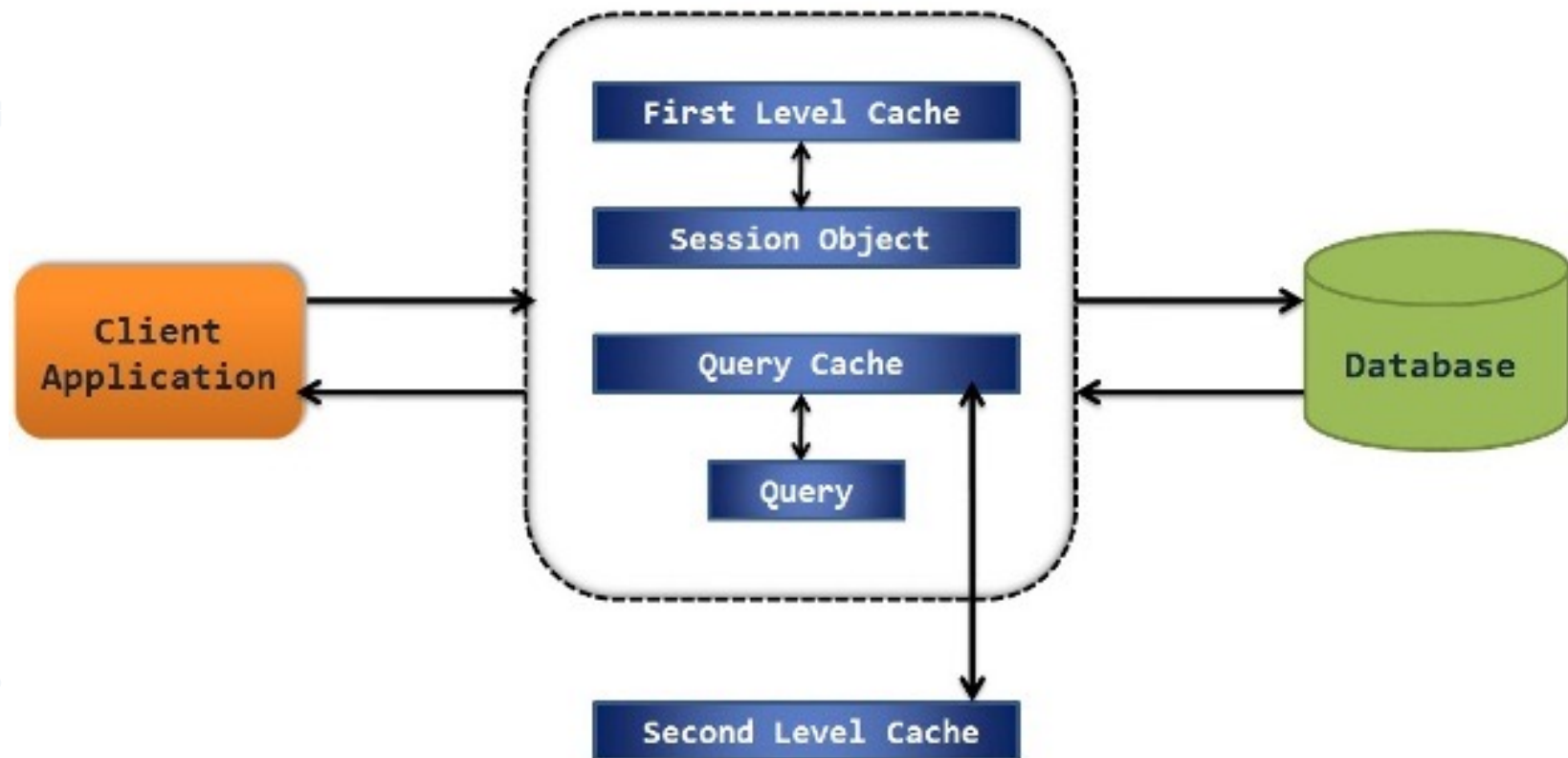
Hibernate cache

Довольно часто в java приложениях с целью снижения нагрузки на БД используют кеш. Не много людей реально понимают как работает кеш под капотом, добавить просто аннотацию не всегда достаточно, нужно понимать как работает система. Поэтому этой статье я попытаюсь раскрыть тему про то, как работает кеш популярного ORM фреймворка. Итак, для начала немного теории.

Прежде всего Hibernate cache — это 3 уровня кеширования:

- Кеш первого уровня (First-level cache);
- Кеш второго уровня (Second-level cache);
- Кеш запросов (Query cache);

Hibernate





Кеш первого уровня

Кеш первого уровня всегда привязан к объекту сессии. Hibernate всегда по умолчанию использует этот кеш и его нельзя отключить. Давайте сразу рассмотрим следующий код:

```
SharedDoc persistedDoc = (SharedDoc) session.load(SharedDoc.class, docId);  
System.out.println(persistedDoc.getName());  
user1.setDoc(persistedDoc);  
  
persistedDoc = (SharedDoc) session.load(SharedDoc.class, docId);  
System.out.println(persistedDoc.getName());  
user2.setDoc(persistedDoc);
```



Кеш запросов


```
Query query = session.createQuery("from SharedDoc doc where doc.name = :name");

SharedDoc persistedDoc = (SharedDoc) query.setParameter("name", "first").uniqueResult();
System.out.println(persistedDoc.getName());
user1.setDoc(persistedDoc);

persistedDoc = (SharedDoc) query.setParameter("name", "first").uniqueResult();
System.out.println(persistedDoc.getName());
user2.setDoc(persistedDoc);
```

Результаты такого рода запросов не сохраняются ни кешом первого, ни второго уровня. Это как раз то место, где можно использовать кеш запросов. Он тоже по умолчанию отключен. Для включения нужно добавить следующую строку в конфигурационный файл:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```




```
Query query = session.createQuery("from SharedDoc doc where doc.name = :name");
query.setCacheable(true);
```


Кеш второго уровня

Если кеш первого уровня привязан к объекту сессии, то кеш второго уровня привязан к объекту-фабрике сессий (Session Factory object). Что как бы подразумевает, что видимость этого кеша гораздо шире кеша первого уровня.

```
Session session = factory.openSession();  
SharedDoc doc = (SharedDoc) session.load(SharedDoc.class, 1L);  
System.out.println(doc.getName());  
session.close();
```

```
session = factory.openSession();  
doc = (SharedDoc) session.load(SharedDoc.class, 1L);  
System.out.println(doc.getName());  
session.close();
```



В данном примере будет выполнено 2 запроса в базу, это связано с тем, что по умолчанию кеш второго уровня отключен. Для включения необходимо добавить следующие строки в Вашем конфигурационном файле


```
<property name="hibernate.cache.provider_class" value="net.sf.ehcache.hibernate.SingletonEhCacheProvider"/>
```

//или в более старых версиях

```
//<property name="hibernate.cache.provider_class" value="org.hibernate.cache.EhCacheProvider"/>  
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

Хибернейт сам не реализует кеширование как таковое. А лишь предоставляет структуру для его реализации, поэтому подключить можно любую реализацию, которая соответствует спецификации нашего ORM фреймворка. Из популярных реализаций можно выделить следующие:

- EHCache
- OSCache
- SwarmCache
- JBoss TreeCache



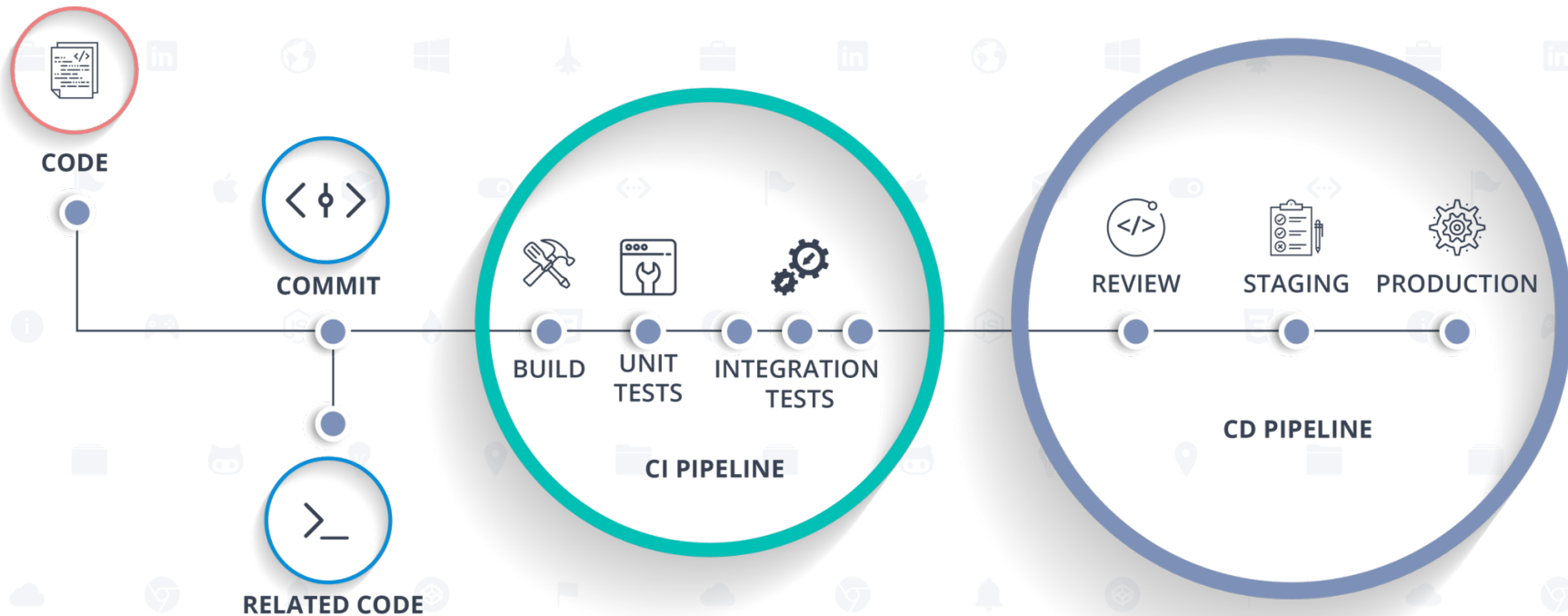
Помимо всего этого, вероятней всего, Вам также понадобится отдельно настроить и саму реализацию кеша. В случае с EHCache это нужно сделать в файле [ehcache.xml](#). Ну и в завершение еще нужно указать самому хибернейту, что именно кешировать. К счастью, это очень легко можно сделать с помощью аннотаций, например так:

```
@Entity
@Table(name = "shared_doc")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class SharedDoc{
    private Set<User> users;
}
```



Стратегии кеширования определяют поведения кеша в определенных ситуациях. Выделяют четыре группы:

- Read-only
- Read-write
- Nonstrict-read-write
- Transactional





Что такое непрерывная интеграция

Непрерывная интеграция (Continuous Integration) в разработке программ это автоматизированный процесс сборки и тестирования кода в разделяемом репозитории. Когда делаются новые коммиты, они изолируются, собираются и тестируются на соответствие определенным стандартам прежде чем волеются в основную кодовую базу.

Непрерывная интеграция позволяет быстро выявлять поломки, ошибки или баги, при этом все перечисленное не попадает в кодовую базу, а исправляется как можно скорее. Команды разработчиков, да и все, желающие стать разработчиками, должны разбираться в том, как работают системы непрерывной интеграции.

Обычно непрерывная интеграция сопряжена с непрерывной доставкой (Continuous Delivery), поэтому этапы непрерывной автоматизированной доставки исполняемого кода в продакшен часто обозначают CI/CD.