МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

О.Литвиненко
"_____"_____2021 р.

ДИПЛОМНИЙ ПРОЄКТ
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
"БАКАЛАВР"

Тема: Онлайнова система управління чергою завдань

Виконавець: Холонівець І.В.

Керівник: Артамонов Є.Б.

Нормоконтролер: Тупота Є.В.

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютеризованих систем управління

Спеціальність 123 "Комп'ютерна інженерія"

(шифр, найменування)

Освітньо професійна програма «Системне програмування»

Форма навчання заочна

ЗАТВЕРДЖУЮ

Завідувач кафедри

О.Литвиненко

«_____»_____2021 р.

ЗАВДАННЯ

на виконання дипломної роботи (проєкту)

Холонівець Іванни Вікторівни

(прізвище, ім'я, по батькові випускника в родовому відмінку)

1.Тема роботи: "Онлайнова система управління чергою завдань "

затверджена наказом ректора від «21» грудня 2020 р. №2523/ст

2. Термін виконання роботи: з 11.01.2021 до 28.02.2021

3. Вихідні дані до проєкту (роботи): постановка задачі до виконання роботи, мови програмування: *PHP*, СУБД: *MySQL*.

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

1) Принципи побудови онлайнової системи упраління чергою завдань та її призначення;

2) Засоби для створення онлайнової системи управління чергою завдань;

3) Реалізація онлайнової системи управління чергою завдань.

5. Перелік обов'язкового графічного матеріалу:

1) Принципи роботи Active MQ;

2) Принципи роботи Kafka;

3) Схема алгоритму системи управління базою даних;

4) Схема алгоритму процесу захоплення завдання;

5) Зв’язок між таблицями бази даних розробленої системи.

6. Календарний план-графік

№ п/п	Етапи виконання дипломного проєкту	Термін виконання етапів	Примітка
1	Провести аналіз літератури за темою дипломнго проєкту та аналіз існуючих систем	11.01.21-12.01.21	
2	Розроблення та затвердження плану дипломного проєкту	14.01.21-18.01.21	
3	Розробка розділу 1	19.01.21-21.01.21	
4	Розробка розділу 2	22.01.21-25.01.21	
5	Розробка розділу 3	26.01.21-01.02.21	
6	Написати пояснювальну записку	02.02.21-14.02.21	
7	Підготувати презентацію	15.02.21-17.02.21	
8	Оформити супрводжувадьну документацію	18.02.21 19.02.21	

7. Дата видачі завдання « 11 » січня 2021 р.

Керівник дипломнго проєкту Артамонов Є.Б.

(підпис)

Завдання прийняв до виконання Холонівець І.В.

(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломного проєкту «Онлайнова система управління чергою завдань»: містить 66 сторінок, 18 рисунків, 11 літературних джерел, 1 додаток.

ЧЕРГА, СИСТЕМА УПРАВЛІННЯ ЧЕРГОЮ, ЧЕРГА ПОВІДОМЛЕНЬ, СЕРВЕР, APACHE, PHP, MYSQL, CRON.

Під час роботи над дипломним проєктом було розроблено програмний засіб для управління чергою завдань.

Об’єкт дослідження – технологія роботи онлайнової системи управління чергою завдань.

Предмет дослідження - існуючі системи обміну повідомленнями.

Мета роботи - розробка власної системи управління чергою завдань мовою програмування PHP.

Метод дослідження - дослідження існуючих рішень для обміну повідомленнями.

Результати дипломного проєкту рекомендується використовувати при розробці нових програмних засобів, які надають можливість працювати з відкладеними завданнями.

Зміст

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ІТ - інформаційні технології

БД - база даних

ОС - операційна система

ПЗ - програмне забезпечення

ООП - об’єктно-орієнтованого програмування

СУБД - система управління базами даних

ВСТУП

Черга - це певна впорядкована послідовність будь-чого чи кого, яка працює за принципом останній прийшов - останній вийшов. Черга потрібна для організації всього, що нас оточує. Без неї світ погряз би у хаосі. Часто черга асоціюється з негативом і втраченими нервами, але без неї все було би куди гірше. Просто уявімо, або краще сказати згадаймо, бо таке мабуть було з кожним, як після двохгодинного очікування прийому лікаря в поліклініці хтось проходить перед вами зі словами “мені тільки запитати”, а сам сидить там пів години. Звісно це не може викликати приємних емоцій.

Але що було би якби поняття черги взагалі не було. Люди є надзвичайно різні і такі ситуації могли би мати найнепередбачуваніші наслідки. Черги всюди і вони слугують певним контролером як в суспільстві так і у всіх сферах нашого життя. Одним із прикладів черги є затори на дорогах. Саме хаосом можна назвати правила дорожнього руху в країнах Азії, а точніше їх відсутність. Водії не дотримуються черг на дорогах і виглядає це досить небезпечно.

Життя без черг, на мою думку, це повернення в кам’яний вік, де виживають найсильніші. З ними - ми можемо розраховувати на певний порядок і впевненість.

Черги в програмуванні використовуються як і в реальному житті, коли потрібно виконати якісь дії в порядку їх надходження. Прикладом може слугувати організація подій в *Windows*. Коли користувач виконує якусь дію з додатком, то в ньому не викликається відповідна процедура (адже в цей момент додаток може виконувати якусь іншу дію), а йому приходить повідомлення з інформацією про виконану дію, це повідомлення ставиться в чергу, і тільки після того, як будуть виконані всі попередні повідомлення, додаток здійснить необхідну операцію.

Черги часто використовуються в програмах для реалізації буфера, в який можна покласти елемент для подальшої обробки, зберігаючи порядок.

Наприклад, якщо база даних підтримує тільки одне з’єднання, можна використати чергу потоків, які будуть чекати своєї черги на доступ до БД.

Черга повідомлень - це деяка система, яка забезпечує збереження і передачу даних між різними учасниками системи. Черги повідомлень практично завжди використовуються у великих системах. Використання систем управління чергою завдань надзвичайно актуальне в сучасному світі. Завдяки таким системам великі системи можуть працювати швидко і надійно, що дуже важливо для користувача.

Які проблеми вирішує система управління чергою завдань:

1. Завдяки використанню черги, компоненти взаємодіють через деякий спільний інтерфейс, але нічого не знають про існування один одного.

2. Економія ресурсів досягається внаслідок можливості розумно розподіляти інформацію, що надходить в чергу від одних процесів, між іншими процесами, що виконують її обробку.

3. Надійність черг досягається завдяки можливості накопичувати повідомлення, амортизуючи нестачу вираховувальних можливостей системи, а також завдяки незалежності компонентів. Крім цього черга може акомодувати збої окремих компонентів, здійснюючи доставку “повідомлень, що запізнились” після відновлення.

4. Гарантія почергової обробки дозволяє точно контролювати потоки даних в системі і запускати асинхронну обробку там, де це необхідно, не хвилюючись, що одна операція виконається раніше іншої, від результату якої вона не залежить.

Метою дипломного проєкту є оцінка та аналіз існуючих систем і як результат реалізація власного програмного рішення, що буде потребувати менше ресурсів, але буде виконувати поставлені перед ним задачі не гірше розглянутих систем.

РОЗДІЛ 1
ПРИНЦИПИ ПОБУДОВИ ОНЛАЙНОВОЇ СИСТЕМИ УПРАВЛІННЯ ЧЕРГОЮ ЗАВДАНЬ І ЇЇ ПРИЗНАЧЕННЯ

1.1 Поняття черги та необхідності роботи з нею.

1.1.1 Загальне поняття.

Черга - це структура даних, яка побудована по принципу *LILO (last in — last out:* останній прийшов - останнім вийшов). В черзі, якщо ви додаєте елемент, який увійшов найпершим, то він вийде теж найпершим. Виходить, що якщо ви додаєте чотири елемента, то перший доданий елемент вийде першим.

Для кращого розуміння роботи черги можна уявити собі чергу в магазині. Ви стоїте посеред неї. Щоб ви опинились біля каси, спочатку потрібно обслужити всіх людей перед вами. А щоб черга дійшла до останньої людини, касир повинен обслужити всіх перед нею (рисунок 1.1)

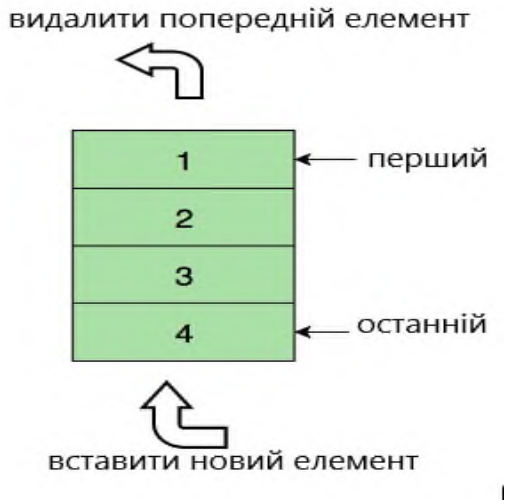

Рис. 1.1. Черга

Черги часто використовуються в програмах для реалізації буфера, в який можна покласти елемент для наступної обробки, зберігаючи порядок надходження. Наприклад, база даних підтримує лише одне з'єднання, можна використати чергу потоків, які будуть, як не дивно, чекати своєї черги на доступ до БД.

1.1.2 Необхідність черг.

Використання черги в програмуванні майже відповідає її ролі в звичайному житті. Черга практично завжди зв'язана з обслуговуванням запитів, в тих випадках, коли вони не можуть бути виконані моментально. Черга підтримує також порядок обслуговування запитів.

Розглянемо, наприклад, що відбувається коли користувач натискає клавішу на клавіатурі комп'ютера. Тим самим він просить комп'ютер виконати якісь дії. Наприклад, якщо користувач просто набирає текст, то дія повинна заключатись в додаванні до тексту одного символа і може супроводжуватися перемальовуванням області екрана, прокруткою вікна, переоформлюванням абзацу та ін.

Будь-яка, навіть найпростіша, операційна система завжди в тій чи іншій мірі багатозадачна. Це означає, що в момент натискання клавіші ОС може бути зайнята якою-небудь іншою роботою. Тим не менш, ОС ні в якому разі не має права проігнорувати натиснуту клавішу. Тому відбувається переривання роботи комп'ютера, він запам'ятовує свій стан і переключається на обробку натискання клавіші. Така обробка повинна бути дуже короткою, щоб не порушити виконання інших задач. Команда, яка віддається натисканням клавіші, просто додається в кінець черги запитів, чекаючих свого виконання. Після цього переривання закінчується, комп'ютер відновлює свій стан і продовжує роботу, яка була перервана натисканням клавіші. Запит, поставлений в чергу, буде виконаний не зразу, а тільки тоді, коли наступить його черга.

В системі *Windows* робота віконних додатків основана на повідомленнях, які посилаються додаткам. Наприклад, бувають повідомлення про натискання

клавіші миші, про закриття вікна, про необхідність перемалювання області вікна, про вибір пункта меню і так далі. Кожна програма має чергу запитів. Коли програма отримує свій квант часу на виконання, вона вибирає черговий запит з початку черги і виконує його. Таким чином, робота віконного додатку складається, спрощено кажучи, з послідовного виконання запитів з її черги. Черга підтримується операційною системою.

Підхід до програмування, що складається не з прямого виклику процедур, а з відправки повідомлень, які ставляться в чергу запитів, має багато переваг і є однією з рис об'єктно-орієнтованого програмування. Так, наприклад, якщо віконній програмі необхідно завершити роботу по якійсь причині, краще не викликати відразу команду завершення, яка небезпечна, тому що порушує логіку роботи і може призвести до втрати даних. Замість цього програма відправляє сама собі повідомлення про необхідність завершення роботи, яке буде поставлене в чергу запитів і виконане після запитів, що поступили раніше.

1.2 Загальне визначення онлайнової системи управління чергою завдань та її призначення.

Система управління чергою завдань - це система, яка перетворює повідомлення по одному протоколу від програми-джерела в повідомлення протоколу програми-приймача, тим самим виступаючи між ними посередником. Крім перетворення повідомлень з одного формату в інший, в завдання системи управління повідомленнями також входить:

1. перевірка повідомлення на помилки;
2. маршрутизація конкретному приймачу (ам);
3. розбиття повідомлення на декілька маленьких, а потім агрегування відповідей приймачів і відправка результату джерелу;
4. збереження повідомлень в базі даних;
5. виклик веб-сервісів;
6. поширення повідомлень передплатникам, якщо використовуються шаблони типу видавець-підписник.

Використання брокерів повідомлень дозволяє розвантажити веб-сервіси в розподіленій системі, так як при відправці повідомлень їм не потрібно витрачати час на деякі ресурсомісткі операції типу маршрутизації і пошуку приймачів. Крім того, система управління повідомленнями для підвищення ефективності може реалізовувати стратегії впорядкованої розсилки і визначення пріоритетності; балансувати навантаження та інше.

1.3 Типи онлайнових систем управління чергою завдань.

Щоб дві системи могли спілкуватись між собою, вони повинні спочатку визначити інтерфейс. Визначення цього інтерфейса включає в себе вибір транспорту чи протоколу, такого як *HTTP*, *MQTT* чи *SMTP* і узгодження форматів повідомлень, якими будуть обмінюватись системи. Це може бути строгий процес, такий як визначення схеми *XML* з вимогами до затрат на корисну нагрузку повідомлення, чи це може бути набагато менш формально, наприклад, угода між двома розробниками про те, що деяка частина *HTTP*-запиту буде містити ідентифікатор клієнта.

Поки формат повідомлень і порядок відправки між системами узгоджені, вони можуть взаємодіяти між собою, не піклуючись про реалізацію іншої системи. Наповнення цих систем, таке як мова програмування чи використовуваний фреймворк, можуть з часом змінитися. До тих пір, поки підтримується сама угода, взаємодія може продовжуватись без змін з другої сторони. Ці дві системи ефективно розділені цим інтерфейсом.

Системи обміну повідомленнями, як правило, передбачають посередника між двома системами, які взаємодіють для подальшого розділення відправника від отримувача. При цьому система обміну повідомленнями дозволяє відправнику відправити повідомлення, не знаючи де знаходиться отримувач, активний він зараз чи скільки їх екземплярів.

Типи систем управління чергою завдань розглянемо на прикладах.

1.3.1 *Point-to-Point.*

Найпростішим прикладом даного типу системи управління чергою завдань може бути відправка посилки поштою. Ви йдете на пошту щоб відправити подарунок мамі. Ви підходите до віконечка і віддаєте його працівнику. Він забирає посилку і видає вам квитанцію. Мамі не потрібно бути вдома в момент відправки вашого подарунка, але ви можете бути впевнені, що вона його отримає в якийсь момент в майбутньому і можете продовжувати займатись своїми справами.

Цей приклад моделі обміну повідомленнями називається точка-точка. Поштове відділення тут виступає механізмом розподілення посилок, даючи гарантію, що кожна посилка буде доставлена один раз. Використання поштового відділення відділяє акт відправки посилки від її доставки.

В класичних системах обміну повідомленнями модель “точка-точка” реалізується через черги. Черга діє як буфер *FIFO* (перший зайшов, перший вийшов), на який може підписатися один або декілька споживачів. Кожне повідомлення доставляється тільки одному з підписаних споживачів. Черги зазвичай стараються справедливо розділити повідомлення між споживачами. Тільки один споживач отримає дане повідомлення.

До черг застосовується термін “надійні” *(«durable»).* Надійність - це якість сервісу, яка гарантує, що система обміну повідомленнями буде зберігати повідомлення при відсутності активних підписників до тих пір, поки споживач не підпишеться на чергу для доставки повідомлень.

Надійність часто плутають з персистентністю, хоча ці два терміни взаємозамінні, вони виконують різні функції. Персистентність визначає чи записує повідомлення система обміну повідомленнями в якого-небудь роду сховище між отриманням і відправкою його споживачу. Повідомлення, які відправляються в чергу, можуть бути або не бути персистентними.

Обмін повідомленнями типу “точка-точка” використовується коли варіант використання потребує однократної дії з повідомленням. В якості прикладу можна привести внесення коштів на рахунок чи виконання замовлення на доставку. Пізніше розкриємо чому система обміну повідомленнями сама по собі

не здатна забезпечити однократну доставку і чому черги можуть в кращому випадку забезпечити гарантію доставки хоча би один раз.

1.3.2 Видавець-Підписник.

Найпростішим прикладом даної моделі може бути онлайн-конференція. Поки ви підключені до конференції, ви чуєте все, що говорить спікер, разом з іншими учасниками. Коли ви відключитесь, то пропустите те, що сказано. При повторному підключенні ви продовжите слухати те, що говорять.

Це приклад публікація-підписка. Конференція виступає як широкомовний механізм. Спікера не хвилює скільки людей в даний момент приєднались до дзвінка - система гарантує, що будь-хто, хто підключиться в даний момент почує про що говориться.

В класичних системах обміну повідомленнями модель обміну повідомленнями “публікація-підписка” реалізується через топіки. Топік представляє такий самий спосіб широкомовлення, як механізм конференц-зв’язку. Коли повідомлення відправляється в топік, воно розподіляється по всім підписаним користувачам.

Топіки зазвичай ненадійні. Як і слухач, який не чує, що говориться на конференції, коли слухач відключається, підписники топіка пропускають будь-які повідомлення, які відправляються в той момент, коли вони знаходяться в автономному режимі. По цій причині можна сказати, що топіки дають гарантію доставки не більше одного разу для кожного споживача.

Обмін повідомленнями типу “публікація-підписка” зазвичай використовується, коли повідомлення носять інформаційний характер, і втрата одного повідомлення не має особливого значення. Наприклад, топік може передавати показники температури від групи датчиків один раз за секунду. Система, яка цікавиться поточною температурою і яка підписується на топік, не буде хвилюватись, якщо пропустить повідомлення - наступне прийде в найближчий час.

1.3.3 Гібридні моделі.

Веб-сайт магазину розміщує повідомлення про замовлення в “чергу повідомлень”. Основним споживачем цих повідомлень є виконавча система. Крім того, система аудиту повинна мати копії цих повідомлень про замовлення для подальшого відстеження. Обидві системи не можуть пропускати повідомлення, навіть якщо самі системи протягом деякого часу недоступні. Веб-сайт не повинен знати про другі системи.

Сценарії використання часто потребують поєднання моделей обміну повідомленнями “публікація-підписка” і “точка-точка”, наприклад, коли декільком системам потрібна копія повідомлення, і для запобігання втрати повідомлення потрібна як надійність, так і персистентність.

В таких випадках потрібен адресат (загальний термін для черг і топіків), який розподіляє повідомлення в основному як топік, так, що кожне повідомлення відправляється в окрему систему, зацікавлену в цих повідомленнях, але і також в якій кожна система може визначити декілька споживачів, які отримують вхідні повідомлення, що більше схоже на чергу. Тип зчитування в цьому випадку - один раз для кожної зацікавленої сторони. Ці гібридні адресати часто потребують надійності, так що якщо споживач відключається, повідомлення, які відправляються в цей час, приймаються після повторного підключення споживачів.

Тепер, коли в нас є деяка база термінів і розуміння того, для чого потрібна система управління чергою завдань, можемо перейти до огляду існуючих систем.

1.4 Огляд існуючих систем.

1.4.1 *ActiveMQ*.

Цю систему управління чергою завдань можна назвати класичною системою обміну повідомленнями. Вона була написана в 2004 році, заповнюючи потребу в брокері повідомлень з відкритим вихідним кодом. В той час, якщо ви хотіли використовувати обмін повідомленнями в своїх системах, єдиним вибором були дорогі комерційні продукти.

ActiveMQ була розроблена як реалізація специфікації *Java Message Service (JMS)*. Це рішення було прийнято, щоб задовольнити вимоги до реалізації *JMS*-сумісного обміну повідомленнями в проєкті *Apache Geronimo* - сервері додатків *J2EE* з відкритим вихідним кодом.

Збереження повідомлень.

Коли брокер отримує персистентні повідомлення, вони спочатку записуються на диск в журнал. Журнал - це структура даних на диску, в яку можна тільки додавати дані і яка складається з декількох файлів. Вхідні повідомлення серіалізуються брокером в незалежне від протокола представлення об’єкта, а потім переводяться в двоїчну форму, яка потім записується в кінець журнала. Журнал містить лог всіх вхідних повідомлень, а також дані про ті повідомлення, які були підтвердженні як прочитані клієнтом.

Дискові адаптери персистентності підтримують індексні файли, які відстежують, де в журналі розміщені наступні повідомлення, що пересилаються. Коли всі повідомлення з файлу журнала будуть прочитані, вони будуть або видалені, або заархівовані фоновим робочим потоком *ActiveMQ*. Якщо цей журнал пошкоджений під час збою брокера, *ActiveMQ* перестроїть його на основі інформації в файлах журналу.

Повідомлення зі всіх черг записуються в одні і ті ж файли журналу, що означає, що якщо одне повідомлення не прочитане, то весь файл не може бути очищеним. Це може з часом викликати проблеми з нехваткою дискового простору.

Класичні брокери повідомлень не призначені для довгострокового збереження.

Журнали є надзвичайно ефективним механізмом для збереження і наступного видобутку повідомлень, оскільки доступ до диска для обох операцій є послідовним.

Відправка повідомлень в чергу.

Рисунок 1.3 показує нам взаємодію, яка відбувається при відправці повідомлень.

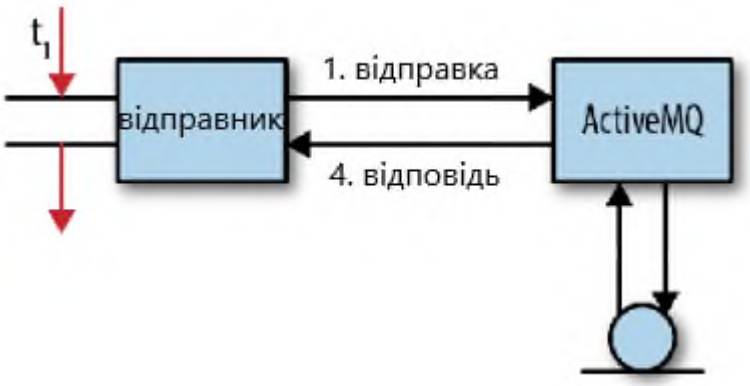

Рис. 1.3. Відправка повідомлень в JMS

В клієнтській системі потік отримує покажчик на *MessageProducer*. На першому етапі відправляючий потік викликає клієнтську бібліотеку і маршалізує повідомлення в формат *OpenWire*. Потім повідомлення відправляється брокеру.

Другий етап - це запис повідомлення в сховище. Після запису, на третьому етапі, персистент-адаптер повинен отримати підтвердження того, що повідомлення дійсно було записане.

Як тільки брокер переконається, що повідомлення збережене, він відправить клієнту відповідь-підтвердження. Це четвертий етап.

Внутрішні конфлікти.

Використання одного журнала для всіх черг додає додаткову складність. В будь-який момент часу може існувати декілька продюсерів, які одночасно відправляють повідомлення. В брокера є декілька потоків, які отримують ці повідомлення. Кожен потік не може одночасно писати в один і той же файл, так як записи будуть конфліктувати один з одним, то записи повинні бути поставлені в чергу з допомогою механізма виключення. Це називається конфлікт потоків.

ActiveMQ включає в себе буфер запису, в який приймаючі потоки записують свої повідомлення, очікуючи завершення попереднього запису. Потім буфер записується в одну дію, коли повідомлення стає доступним. Після завершення потоки отримують сповіщення. Таким чином, брокер максимізує використання пропускної здатності сховища.

Вичитування повідомлень з черги.

Процес вичитування повідомлень починається тоді, коли споживач готовий їх прийняти. Це можна побачити на рисунку 1.4.

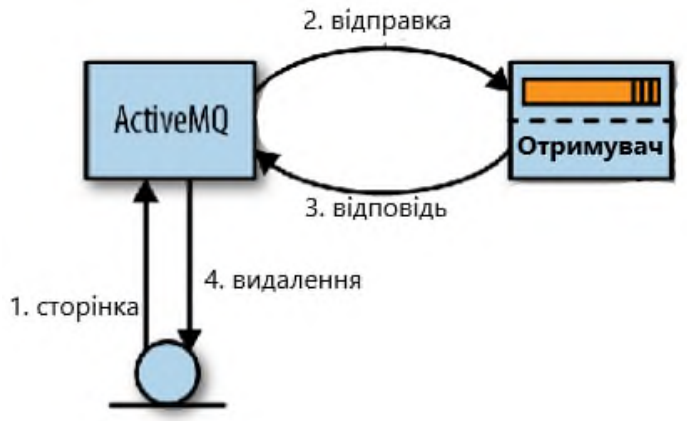

Рис. 1.4. Вичитування повідомлень за допомогою *JMS*

Коли *ActiveMQ* стає відомо про консюмер, він посторінково читає повідомлення зі сховища в пам’ять для розповсюдження. Це етап один. На другому етапі ці повідомлення перенаправляються отримувачу, часто частинами для зниження об’єму мережевої взаємодії. Брокер відстежує, які повідомлення були направлені і якому отримувачу.

На третьому етапі брокеру відправляється підтвердження про вичитку. Як тільки брокер приймає підтвердження доставки повідомлення, воно видаляється зі сховища повідомлень. Це етап чотири.

1.4.2 *Kafka*.

В *Kafka* прийнята архітектура, яка перевизначила ролі і обов'язки клієнтів і брокерів обміну повідомленнями. Модель *JMS* дуже орієнтована на брокер, де він відповідає за розповсюдження повідомлень, а клієнти повинні піклуватися тільки про відправку і отримання повідомлень.

Kafka орієнтована на клієнта, при цьому клієнт бере на себе багато функцій традиційного брокера, такі як справедливе розподілення відповідних повідомлень серед споживачів, в обмін отримуючи надзвичайно швидкий і масштабований брокер.

Уніфікована модель адресата.

Kafka об'єднала обмін повідомленнями типу "публікація-підписка" і "точка-точка" в рамках одного виду адресата - топіка. Це збиває з толку людей, які працювали з системами обміну повідомленнями, де слово "топік" відноситься до механізму, з якого (з топіка) читання не є надійним. Топіки *Kafka* варто розглядати як гібридний тип адресата.

В кожного топіка *Kafka* є свій журнал. Продюсери, що відправляють повідомлення в *Kafka*, дописують в цей журнал, а отримувачі читають з журнала з допомогою покажчиків, які постійно переміщуються вперед. Періодично *Kafka* видаляє найстаріші частини журнала, незалежно від того, чи були повідомлення в цих частинах прочитані. В *Kafka* брокер не піклується про те, прочитані повідомлення чи ні - це відповідальність клієнта.

Ця модель повністю відрізняється від *ActiveMQ*, де повідомлення зі всіх черг зберігаються в одному журналі, а брокер помічає повідомлення як видалені після того як вони були прочитані.

Журнал *Kafka* складається з кількох розділів (рисунок 1.5). *Kafka* гарантує чіткий порядок кожного розділу. Це означає, що повідомлення, записані в розділ

в конкретному порядку, будуть прочитані в такому ж. Кожен розділ реалізований у вигляді циклічного файла журналу, який складається з підмножини всіх повідомлень, відправлених в топік його продюсерами. Створений топік включає за замовчуванням один розділ. Ідея розділів - це центральна ідея *Kafka*.

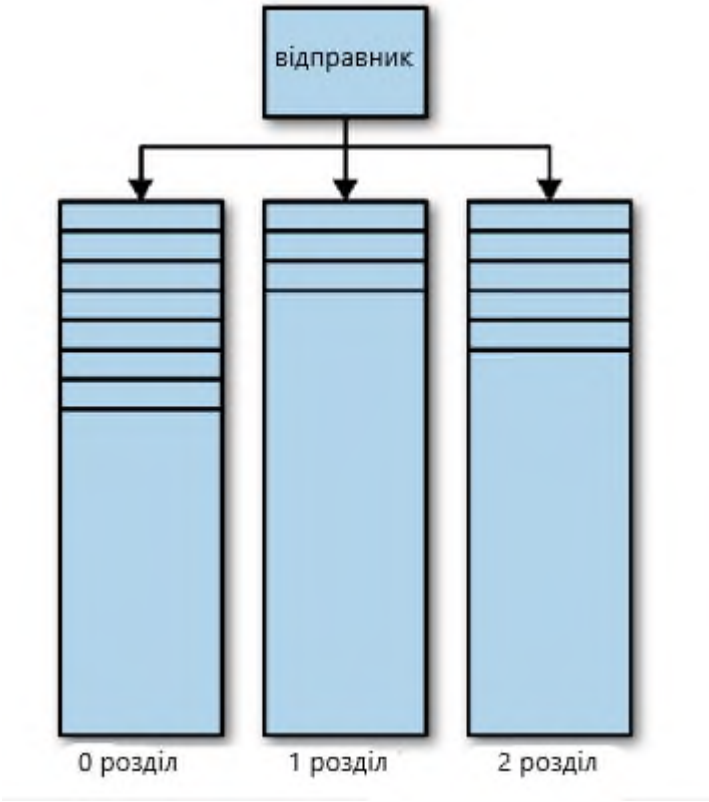

Рис. 1.5. Розділи Kafka

Читання повідомлень.

Клієнт, який хоче прочитати повідомлення, керує іменованим покажчиком, який називається групою споживачів, який вказує на зміщення повідомлення в розділі. Зміщення - це позиція зі зростаючим номером, яка починається з 0 на початку розділу. Ця група споживачів, на яку посилаються в *API* через ідентифікатор *group_id*, який визначається користувачем, відповідає одному логічному отримувачу чи системі.

На рисунку 1.6 показано топік з одним розділом.

Рис. 1.6. Отримувач читає з розділу

Коли отримувач підключається зі своїм *group_id* до цього топіка, йому призначається розділ для читання і зміщення в цьому розділі. Положення цього зміщення конфігурується в клієнті, як покажчик на найновіше повідомлення або найстаріше повідомлення. Отримувач робить запит на повідомлення з топіка, що призводить до їх послідовного читання з журналу.

Позиція зміщення регулярно комітиться назад в *Kafka* і зберігається, як повідомлення у внутрішньому топіку. Прочитані повідомлення не видаляються, на відміну від звичайного брокера, і клієнт може премотати зміщення, щоб повторно обробити вже прочитані повідомлення.

Коли підключається другий логічний отримувач, використовуючи другий *group_id*, він керує другим покажчиком, який не залежить від першого (рисунок 1.7). Таким чином, топік *Kafka* діє як черга, в якій існує один отримувач, і як звичайний топік “видавець-підписник”, на який підписані декілька споживачів, з додатковою перевагою, що всі повідомлення зберігаються і можуть оброблятись декілька раз.

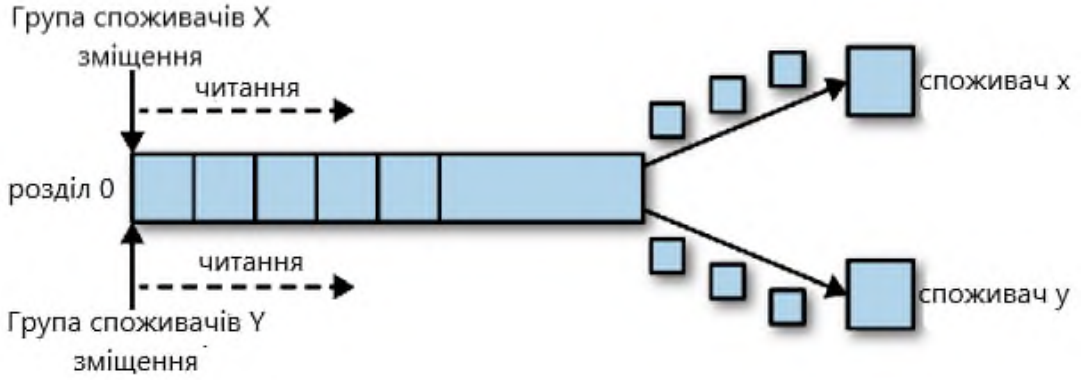

Рис.1.7. Два отримувач в різних групах споживачів читають з одного розділу

Якщо декілька отримувачів підключились з одним і тим же *group_id* до топіка з одним розділом, то отримувач, який підключився останнім, буде контролювати покажчик з цього моменту і буде отримувати всі повідомлення (рисунок 1.8).

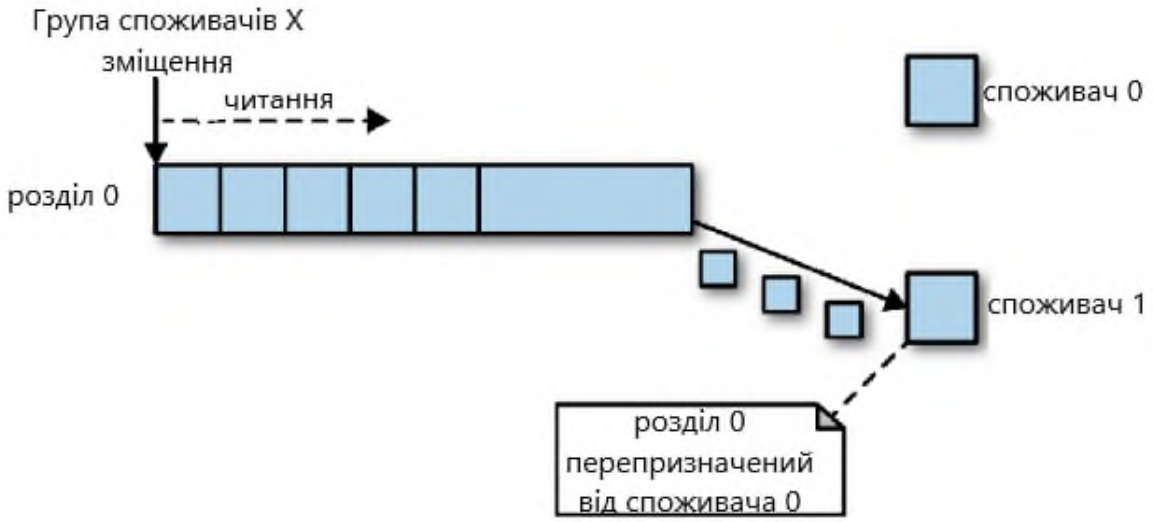

Рис. 1.8. Два отримувача в одній і тій же групі споживачів читають з одного розділу

Цей режим обробки, в якому кількість отримувачів перевищує число розділів, можна розглядати як різновид монопольного споживача.

На рисунку 1.9 показаний сценарій, де споживачу надається контроль над покажчиками, які відповідають його *group_id* в обох розділах, і починається читання повідомлень з обох розділів.

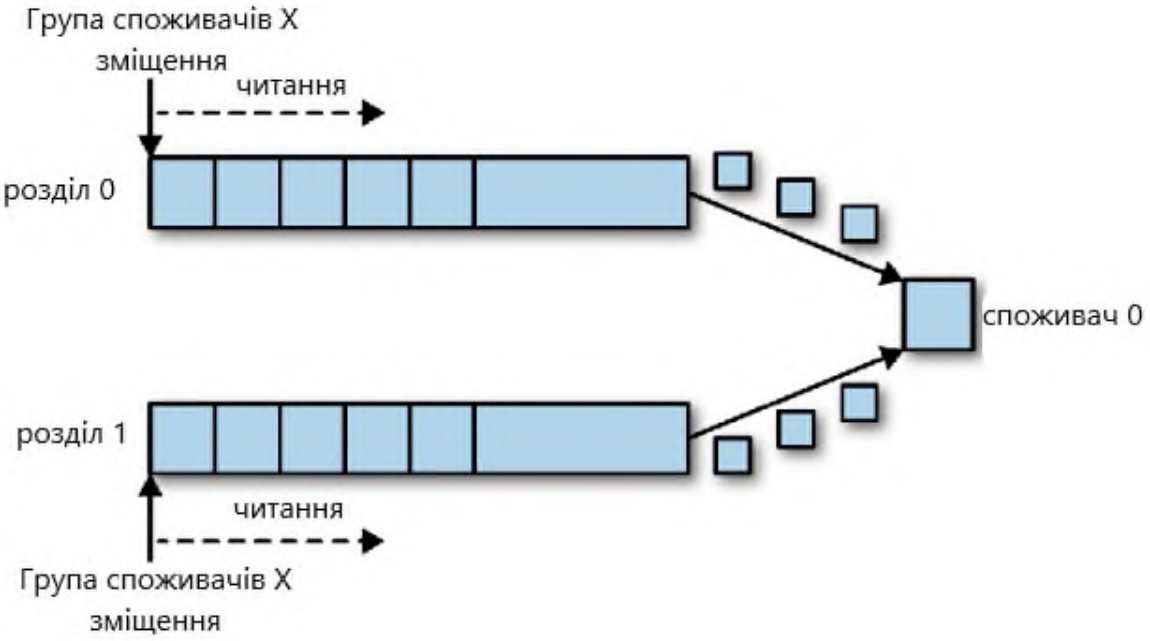

Рис. 1.9. Один отримувач читає з декількох розділів

Коли в топік додається додатковий отримувач для того ж *group_id, Kafka* перепризначає один з розділів з першого на другий отримувач. Після цього кожен отримувач буде вичитувати з одного розділу топіка (рисунок 1.10).

Щоб забезпечити обробку повідомлень паралельно в 20 потоків, вам буде потрібно як мінімум 20 розділів. Якщо розділів буде менше, у вас залишаться отримувачі, яким не буде над чим працювати, що описано раніше при обговорені монопольних споживачів.

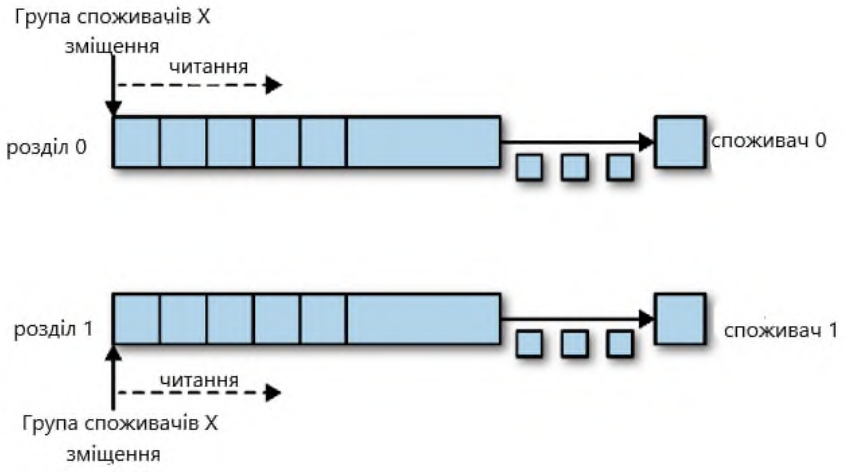

Рис. 1.10. Два отримувача в одній і тій же групі споживачів читають з різних розділів

Ця схема значно знижує складність роботи брокера *Kafka* в порівнянні з розподіленням повідомлень, необхідним для підтримки черги *JMS*. Все, що повинен зробити брокер *Kafka* - це почергово передавати повідомлення споживачу, коли останній робить на них запит.

Відправка повідомлень.

В той час, як в *JMS* ми використовуємо структуру повідомлення з метаданими (заголовками і властивостями) і тілом, які містять корисну нагрузку, в *Kafka* повідомлення є парою “ключ-значення”. Корисна нагрузка повідомлення відправляється, як значення. Ключ, з іншої сторони, використовується головним чином для партиціонування і повинен містити специфічний для бізнес-логіки ключ, щоб помістити зв’язані повідомлення в той же розділ.

Висновок до розділу 1.

У даному розділі було визначене загальне поняття онлайнової системи управління чергою завдань та її призначення. У більш глибокому ознайомленні було визначено типи, на які поділяються ці системи: точка-точка, видавець-підписник, гібридні моделі. На прикладах викладено відмінності між ними.

Крім загального визначення було розглянуто існуючі системи на прикладі *ActiveMQ* і *Kafka*. Ці системи мають суттєві розбіжності. В цьому розділі було описано основні моменти в їх роботі, такі як збереження, читання, відправка повідомлень, та інші важливі деталі.

Ці системи працюють по типу видавець-підписник та гібридна модель відповідно. В наступному розділі буде розглянуто засоби для створення власної системи управління чергою завдань, яка буде працювати по типу точка-точка.

РОЗДІЛ 2
ЗАСОБИ ДЛЯ СТВОРЕННЯ ОНЛАЙНОВОЇ СИСТЕМИ УПАРВЛІННЯ ЧЕРГОЮ ЗАВДАНЬ

2.1 Сервер.

Сервер - це спеціальне обладнання (зазвичай службовий комп’ютер чи робоча станція), яке покликане виконувати сервісне програмне забезпечення без участі людини. В перекладі з англійської, *serve* - служити, а *server* - виконуючий службу, тобто службове обладнання або програмне забезпечення.

Головне правило сервера, яке відрізняє його від звичайного комп’ютера, - це автономність. Тобто участь людини не вимагається. Людина всього лиш здійснює первісне налаштування, періодичне апаратно-технічне обслуговування і обслуговування в нештатних ситуаціях.

Якщо говорити людською мовою, то сервер - це комп’ютер, який обробляє команди користувачів для їх коректного виконання. Це деякий проміжний етап між діями користувача і відповіддю програми на ці дії.

2.1.1 Загальні поняття.

В якості сервера може виступати як окремий службовий комп’ютер (схожий на звичайний ПК), так і ціла станція, яка включає в себе безліч апаратних одиниць. Часто для виконання внутрішніх задач компанії (наприклад, роботи з базами даних, обчислювальних операції, відправки і отримання внутрішньої пошти) використовуються одиночні службові комп’ютери (рисунок 2.1).

Рис. 2.1. Службовий комп’ютер

Якщо ж потужностей одного сервера не вистачає і необхідно підключати додаткові, використовують станції, які представляють собою вертикальні стойки з множиною апаратних одиниць. Наприклад, у великих компаніях, дата-центрах чи у хостинг-провайдерів (рисунок 2.2).

Рис. 2.2. Серверна станція

Такі компанії мають окремі серверні кімнати, де і розміщується все обладнання. В приміщеннях підтримується певний рівень вологості, температури, запиленості, передбачені протипожежні заходи безпеки і т. д.

Серверні станції можуть використовуватись як для виконання задач всередині компанії, так і для здачі сервера в оренду. Наприклад для розміщення сайтів.

По суті, сервер - це той же комп’ютер, тільки з більш якісними апаратними складовими (оперативна пам’ять, процесори, жорсткі диски і т. д.). Пристрої вводу і виводу (монітор, клавіатура, мишка) для виконання операцій не потрібні, вони потрібні лише для налаштування і обслуговування. Для коректної роботи

необхідні постійне електропостачання, доступ до мережі і справність обладнання.

Алгоритм роботи сервера (рисунок 2.3):

1. Спочатку користувач дає запит, виконує команду (найчастіше це натискання тої чи іншої кнопки).

2. Інформація про його дію приходить на сервер і обробляється обладнанням.

3. Після цього, у співвідношенні з налаштуваннями програми, система виводить ту, чи іншу інформацію на монітор.

Наприклад, користувач вводить логін і пароль на сайті, а потім натискає кнопку “Увійти”. Інформація приходить на сервер, де перевіряється правильність введених даних. Якщо вони введені невірно, користувач побачить повідомлення про помилку. Якщо дані вказані коректно, користувач потрапить у свій “Особистий кабінет”.

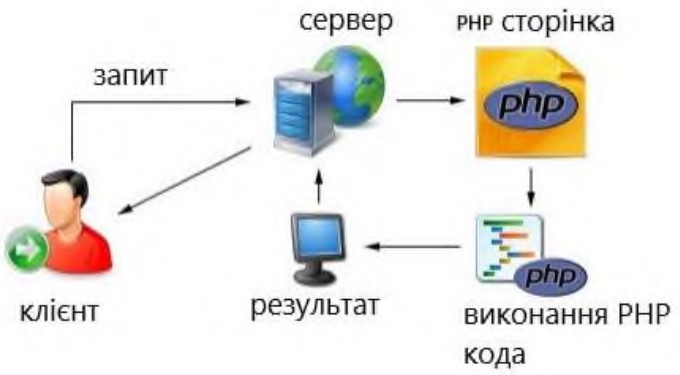

Рис. 2.3. Алгоритм роботи сервера

Головне призначення серверів - це надання доступу до інформації і програмам третім лицям. Наприклад, надання співробітникам компанії доступу до *CRM*-системи. Або можливість спільної роботи і взаємодія працівників з різних філіалів (міст, країн). АКбо взаємодія користувачів з інтернет-ресурсом, онлайн-грою, додатком, базою даних і т. д. Для всього цього необхідні сервери.

Всі сервери діляться на декілька видів в залежності від того, яку задачу вони покликані виконувати.

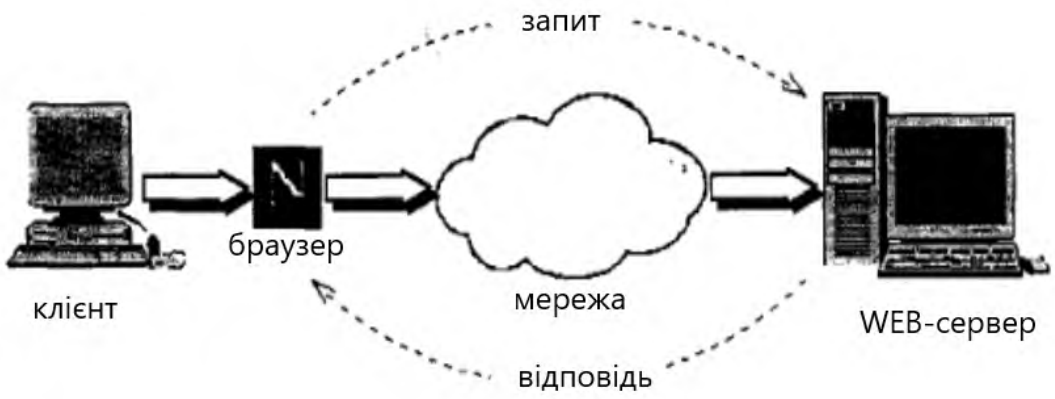

Рис. 2.4. *WEB*-сервер

WEB-сервер (рисунок 2.4) використовується для надання загального доступу до сайта. Таке обладнання повинне мати постійний доступ до інтернету, щоб в будь-який час доби користувач з будь-якої точки світу зміг зайти на веб-ресурс. Всі сайти, розміщені в мережі, використовують онлайн-сервери, які приймають запити користувачів по протоколу *HTTP* і є провідниками між користувачем і ресурсом.

Рис. 2.5. Поштовий сервер

Поштовий сервер (рисунок 2.5) відповідає за відправку електронних повідомлень між користувачами. Відправник пише лист і натискає кнопку “Відправити”. Інформація потрапляє на поштовий сервер, де індексується адреса отримувача. Після цього повідомлення відправляється на потрібну адресу і отримувач може його прочитати. Поштові сервери бувають різні і можуть взаємодіяти між собою, обмінюючись інформацією.

Рис. 2.6. Файловий сервер

Файловий сервер (рисунок 2.6) використовується для надання доступу до файлів мережі іншим комп'ютерам. Тобто люди з різних ПК можуть обмінюватись між собою файлами (загружати, скачувати) або отримувати доступ до даних, що зберігаються на сервері. В таких ситуаціях серверне обладнання повинно мати достатньо місця на жорсткому диску для збереження документів, зображень, відео і т. д. Для обміну файлами використовується протокол *FTP*.

Майже всі програми використовують бази даних. Наприклад, для збереження інформації про всі акаунти (персональні дані, логіни, паролі і т. д.). Для цього використовуються сервери баз даних. Вони виступають в ролі сховища. Для коректної обробки деяких запитів користувача (наприклад авторизація на сайті) потрібно звірити дані з наявними в базі. В залежності від відповідності/невідповідності система дає ту чи іншу відповідь.

Рис. 2.7. Проксі-сервер

Проксі-сервер (рисунок 2.7) це той же онлайн-сервер, тільки виступає він в ролі посередника між користувачем і кінцевим сервером. Тобто запит

користувача буде виконаний не напряму, а через проксі-сервер. При цьому інформація може бути замінена (як сам запит, так і відповідь). Використання проксі-серверів дозволяє зберігати анонімність користувача, захистити комп'ютер від атак і т. д.

Принт-сервер - це програмне забезпечення або обладнання, яке дозволяє використовувати один принтер декільком користувачам разом. Наприклад може бути декілька комп'ютерів, підключених до одного принтера в офісі.

Ігровий сервер використовується компаніями, які займаються розробкою і підтримкою онлайн-ігр. Обладнання повинне витримати нагрузку, коли в грі одночасно знаходяться сотні, тисячі і навіть мільйони гравців. Всі дії гравців і обмін інформацією виконується через ігрові сервера. В онлайн-іграх важлива роль відводиться швидкості обробки інформації, щоб взаємодія гравців один з одним виконувалась в реальному часі.

Крім розділення по типу виконуваних задач, сервери також діляться на дві групи: віддалені і локальні. Назви говорять самі за себе. Віддалений сервер - це обладнання, доступ до якого здійснюється віддалено (наприклад, через інтернет). Тобто у власника немає доступу до самого заліза. Локальний - серверне обладнання до якого є безпосередній доступ.

Прикладом віддаленого сервера є хостинг сайта, який купується у хостинг-провайдера. Власник ресурса не має безпосереднього доступу до обладнання, а може лише керувати хостингом віддалено. Це зручно, так як не приходиться хвилюватись про збереження і працездатність заліза. При цьому вартість його використання досить низька. Приклад локального - сервер, розміщений на території компанії і виконуючий внутрішні задачі.

Сервер - це обладнання, яке обробляє запити користувачів і дає на них відповідь. Натискання кожної кнопки на сайті задіює серверне обладнання. Це важливий елемент, без якого не було би ні інтернету, ні програм, що працюють з різною інформацією, ні можливості взаємодіяти з іншими користувачами.

2.1.2 ОС *Linux*.

Linux - це операційна система, яка на сьогоднішній день є фактично єдиною альтернативою *Windows*. Операційна система складається з декількох основних програм, які потрібні вашому комп'ютеру, щоб він міг спілкуватись і отримувати інструкції від користувачів, читати і записувати дані на жорсткі диски, контролювати використання пам'яті, запускати інші програми. Найважливіша частина ОС - ядро. В системі *GNU/Linux* ядром є *Linux*. Частина, що залишилася складається з інших програм, багато з яких написані проєктом *GNU* або для проєкта *GNU*. Оскільки одне ядро *Linux* - це ще не вся працююча операційна система, повна її назва звучить *GNU/Linux*.

Linux створена по зразку операційної системи *Unix*. З самого початку *Linux* розроблялась як багатозадачна, багатокористувацька система. Цих факторів достатньо щоб зробити *Linux* відмінною від інших широко відомих ОС. Однак, *Linux* відрізняється набагато більше, ніж може здатись на перший погляд. В першу чергу - вона нікому не належить. Значна частина системи розроблена добровольцями безкоштовно.

Розробка того, що пізніше стане *GNU/Linux* почалась в 1984 році, коли Фонд Вільного Програмного забезпечення почав розробку вільної *Unix*-подібної операційної системи *GNU*.

Проєкт *GNU* розробив всеосяжний набір вільного програмного забезпечення для використання в *Unix* і *Unix*-подібних операційних системах, таких як *Linux*.

Хоча багато робочих груп і окремі люди вклали свою працю у створення *Linux*, найбільшим поки ще залишається вклад Фонду Вільного Програмного Забезпечення. Крім розробки більшості інструментів, фонд також створив філософію і організував спільноту (незалежних програмістів і ентузіастів-користувачів). Завдяки цьому в свою чергу, стала можливою поява налаштованих *Linux* систем.

Ядро *Linux* вперше з'явилось в 1991 році, коли фінський студент Лінус Торвальдс анонсував першу заміну ядра *Minix*. Система була названа на честь

його імені. Розквіт популярності *Linux* почався з самого її виникнення. Це пов'язано, в першу чергу, з тим, що ядро цієї ОС, як і більшість програм, написаних під неї, мають дуже важливі якості.

1. Безкоштовність. Можливо, декілька років назад це питання було не так актуальне, але зараз до інтелектуальної власності відношення інше. Все більше людей розуміють, що піратська копія *Windows* може принести великі неприємності. А на платну ліцензійну версію розщедритись мало хто готовий. Так як і на покупку програм, працюючих під даною ОС. Встановивши *Linux*, можна отримати набір із тисяч безкоштовних програм. Хоч вони і не настільки звичні як *Windows*-програми, але абсолютно функціональні.

2. Надійність. Коректна робота апаратної частини вашого ПК, дозволить Linux працювати роками без перезагрузки і зависань.

3. Безпечність. В *Linux* практично немає вірусів. Сама побудова операційної системи виключає роботу шкідливих програм. І тому можна обійтись без антивірусів, які до речі теж коштують не мало і тормозять роботу комп’ютера.

4. Відкритий вихідний код. Це дає можливість використовувати і модифікувати код по своєму бажанню. Можна в будь-який момент виправити які-небудь помилки чи недоліки системи, а також розширити її функціональність, шляхом написання доповнень чи програм, працюючих під її керівництвом.

В наш час навколо Лінукс сформувалась велика кількість програмістів, які постійно вдосконалюють систему. Вони розробляють нові версії і різновиди даної ОС, пишуть найрізноманітніші програми, що працюють під Linux. Найсильніша сторона цієї операційної системи - серверне обслуговування.

2.2 Програмне забезпечення на сервері.

2.2.1 *Apache*.

Роботу інтернету неможливо уявити без веб-сервера. Він відповідає за передачу даних від фізичного сервера, де розміщені сайти чи другі веб-ресурси, на комп'ютери користувачів.

Веб-сервер працює як гігантська віртуальна служба доставки. Він перетворює введені в браузер запити в *IP*-адреси конкретних сховищ, звідки доставляє необхідний контент до його кінцевих споживачів.

Звання найпопулярнішого веб-сервера в світі вже більше 25 років утримує за собою *Apache HTTP Server*, який прийнято називати скорочено *Apache* або "Апач". Сьогодні програма обслуговує більше 40% всіх існуючих серверів, включаючи проєкти *IBM*, *eBay*, *PayPal* і *Facebook*.

Apache - це відкрите програмне забезпечення для створення веб-сервера. Його головна функція - швидка і надійна доставка контента в мережі Інтернет. Веб-сервер приймає запити від клієнтів через веб-браузер по протоколу *HTTP*/*HTTPS*. У відповідь Apache відправляє браузеру контент в вигляді статичних *HTML*-сторінок.

Apache *HTTP Server* був випущений в 1995 році розробником Робертом Макколом з Університету штату Іллінойс. Продукт виник як доопрацьована версія іншого *HTTP*-клієнта - *NCSA HTTPd* 1.3, створеного Робертом раніше.

Основою для модифікації стали багаточисленні "патчі" або програмні "заплатки" для *NCSA*. Саме звідси (а не від індійського племені апачів) походить назва *Apache*. Вона розшифровується як "*a patchy server*" або "сервер з патчами".

Розробкою і підтримкою продукта з 1999 року займається організація *Apache Software Foundation (ASF)* - спільнота експертів-ентузіастів зі всього світу. Цим же некомерційним фондом була створена офіційна ліцензія ПЗ - *Apache License*.

В 2000 році *ASF* представило нову версію *Apache* 2.0 з повністю переробленою архітектурою, вільною від кода *NCSA*. З цього моменту веб-сервер розвивається по двох основних гілках - 1.*x* і 2.*x*.

Apache складається з ядра і динамічної модульної системи. Параметри системи змінюються з допомогою конфігураційних файлів. Для запуску на сервері декількох веб-проєктів одночасно використовується механізм віртуальних хостів.

Ядро Apache розроблено *Apache Software Foundation* на мові *C*. Основні функції - обробка конфігураційних файлів, протокол *HTTP/HTTPS* і загрузка модулів. Ядро може працювати без модулів, але буде мати обмежений функціонал.

Модуль - окремий файл, підключення якого розширює початковий функціонал ядра. Вони можуть включатись в склад ПЗ при початковій установці чи підгружатись пізніше через зміну конфігураційного файлу.

Більшість з них відповідає за конкретний аспект обробки клієнтського запиту - підтримку різних мов програмування, безпечність, кешування, аутентифікацію і т. д. Таким чином, більша задача розбивається на декілька фаз, кожну з яких вирішує окремий вузькоспеціалізований модуль.

Для *Apache* існує більше 500 модулів. Багато популярних веб-додатків зразу випускаються у вигляді модуля для *Apache*.

Система конфігурації Apache працює на текстових файлах з прописними настройками. Вона підрозділяється на три умовних рівня, для кожного з яких є свій конфігураційний файл:

1. Рівень конфігурації сервера (файл *httpd.conf*) - основний конфігураційний файл. Дія розповсюджується на весь механізм веб-сервера.
2. Рівень каталога (файл *.htaccess*) - додатковий конфігураційний файл. Його директиви охоплюють тільки каталог, де розміщений файл, а аткож вкладені підкаталоги.
3. Рівень віртуального хоста (файл *httpd.conf* або *extra/httpd-vhosts.conf*)

Зазвичай конфігураційні файли *Apache* знаходяться в папці "*conf*", а додаткові конфігураційні файли у вкладеній в неї папці "*extra*". Внести зміни можна як через редагування самого файла, так і через командну строку.

Веб-хост - це компонент сервера, що відповідає за обслуговування одного розміщеного на ньому об'єкту (сайта, віртуального сервера). Система

віртуальних хостів *Apache* дозволяє одночасно запускати декілька проєктів з однієї *IP*-адреси.

В *Apache* можна встановити настройки модуля і ядра, а також вводити ліміти на використання серверних ресурсів (трафік, *RAM, CPU*) для кожного віртуального хоста окремо. Це технологічна основа всього механізму веб-хостинга.

2.2.2 *PHP*.

PHP - це популярна мова програмування, особливо у сфері веб-розробників. Автором початкової версії є Расмус Лердорф, ідея якого полягає в розробці набору інструментів для спрощення процесу створення динамічних веб-сторінок. Не дивлячись на те, що сучасний *PHP* є мовою загального призначення, його частіше всього використовують як серверний інструмент для генерації *HTML*-коду, який потім інтерпретується браузером.

PHP - це мова програмування з відкритим вихідним кодом, над розвитком якого працюють програмісти-ентузіасти зі всього світу. Він має простий синтаксис, частково схожий на *Java* і *C++*. Це проєкт, який постійно розвивається. На даний момент актуальною є 8-ма версія мови. По статистиці, кожен шостий програмний продукт створений на *PHP*.

На сьогоднішній день є три основні області використання *PHP*:

1. Для написання скриптів і повноцінних веб-додатків, які виконуються на серверній стороні. Це найпопулярніша сфера використання, оскільки мова початково створювалась саме для веб-розробок. Для повноцінної роботи веб-додатків, написаних на *PHP*, необхідні сервер, парсер і клієнтське ПЗ (веб-браузер), яке відображає результат виконання кода.
2. Для створення сценаріїв, які виконуються в командній строці. Такі міні-додатки можуть працювати на любому ПК. Для їх виконання потрібен тільки парсер. Поскільки *PHP* включає потужні інструменти для роботи зі строками, такі сценарії найчастіше створюють для обробки текстових даних.

3. Для написання графічних інтерфейсів. *PHP* має багато розгалужень, які створені для реалізації різноманітних задач. Одним із таких розгалужень є *PHP-GTK*. Його зазвичай використовують ті програмісти, які звикли до синтаксису *PHP*.

Популярність *PHP* обумовлена його наступними перевагами:

1. Простий і інтуїтивно зрозумілий синтаксис. Він увібрав в себе особливості таких популярних мов програмування, як *C, Java* і *Perl*. Код *PHP* легко читається незалежно від способу використання (для написання невеликих скриптів чи створення потужних додатків з використанням об'єктно-орієнтованого підходу до реалізації програми).

2. Кросплатформеність і гнучкість. *PHP* сумісний з усіма популярними платформами (*Linux, Windows, MacOS*). Написані на ньому додатки успішно працюють на різних серверних ПЗ.

3. Чудова масштабованість. *PHP* дозволяє добитись максимальної продуктивності додатків, написаних на ньому, з ростом апаратних ресурсів. Веб-додатки, розподілені на декілька серверів, здатні впоратись з істотними нагрузками (великим трафіком).

4. Вбудованість в *HTML*-документи. На просту *HTML*-сторінку можна легко додати контент, який буде динамічно змінюватись шляхом вставки блоків коду *PHP*. Вони додаються подібно *HTML*-тегам, не порушують структуру документа.

5. Активний розвиток і удосконалення. Спільнота розробників постійно трудиться над впровадженням додаткового функціоналу, який розширює можливості мови, спрощенням синтаксису і покращенням захисту від можливих атак.

6. Детальна документація. На офіційному сайті проєкту представлені повні відомості про кожну функціональну одиницю мови з прикладом використання.

7. Простий пошук рішень виникаючих проблем. В інтернеті існує велике число форумів, присвячених програмуванню на *PHP*.

8. Широкі перспективи подальшого розвитку. Більшість *CMS* були створені на чистому *PHP* і фреймворках.

2.2.3 *MySQL*.

MySQL представляє собою одну з найрозповсюдженіших на сьогодні систем управління базами даних в мережі Інтернет (рисунок 2.8). Дана система використовується для роботи з досить великими об'ємами інформації. Однак *MySQL* ідеально підходить як для невеликих, так і для масштабних інтернет-проєктів. Немало важливим фактором є безкоштовність системи.

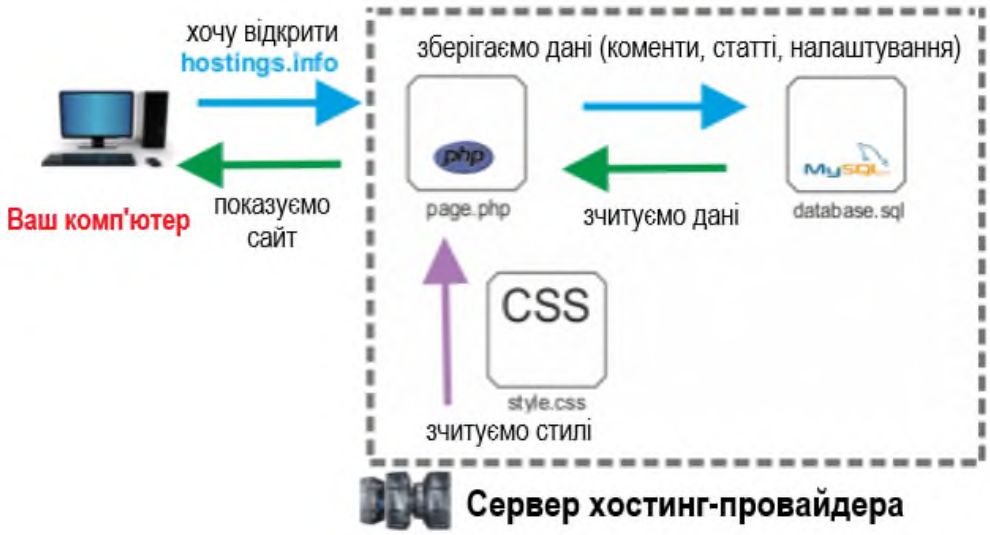

Рис. 2.8. СУБД

Коли користувач старається відкрити сторінку сайту (*page.php*), то перед тим як він побачить сайт, на сервері хостинг-провайдера відбудеться наступне:

1. Виконається *PHP*-код з файлу *page.php*
2. З бази даних (*database.sql*) буде зчитано весь текстовий контент сторінки
3. З файлу стилів (*style.css*) будуть зчитані стилі
4. Користувачу буде показана сторінка, яку він хотів побачити

Також потрібно розуміти, що користувач може залишати коментарі на сторінці, додавати статті і багато іншого. В цей час, всі зміни зберігаються в базу даних, і коли сторінка буде запрошена наступного разу - вона вже буде оновленою (так як з бази даних зчитується оновлена інформація).

База даних представляє собою структуровану сукупність даних. Ці дані можуть бути будь-якими - від простого списка майбутніх покупок до переліку експонатів картинної галереї чи величезної кількості інформації корпоративної мережі. Для запису, вибірки і обробки даних, що зберігаються в комп'ютерній базі даних, необхідна система управління базою даних, якою і є програмне забезпечення *MySQL*. Оскільки комп'ютери прекрасно справляються з обробкою великих об'ємів даних, управління базами даних грає центральну роль у розрахунках. Реалізоване таке управління може бути по-різному - як у вигляді окремих утиліт, так і у вигляді коду, що входить у склад інших додатків.

В реляційній базі даних дані зберігаються в окремих таблицях, завдяки чому досягається вища швидкість і гнучкість. Таблиці зв'язуються між собою з допомогою відношень, завдяки чому забезпечується можливість об'єднати, при виконанні запиту, дані з декількох таблиць. *SQL* як частину системи *MySQL* можна охарактеризувати як мову структурованих запитів.

2.2.4 Планувальник завдань *Cron* та команда *crontab*.

Cron - це демон планування, який виконує задачі з заданими інтервалами. Ці задачі називаються завданнями cron і в основному використовуються для автоматизації обслуговування чи адміністрування системи.

Наприклад, ви можете встановити завдання cron для автоматизації задач, які повторюються, таких як резервне копіювання баз даних чи даних, оновлення системи останніми оновленнями безпеки, перевірка використання дискового простору, відправка електронних листів, перезагрузка сервера і так далі. В деяких додатках, таких як *Drupal* чи *Magento*, для виконання деяких задач потрібні завдання cron.

Завдання cron можуть бути запланованими по хвилині, годині, дню місяця, місяцю, дню тижня чи любій їх комбінації.

Файл *crontab* (таблиця *cron*) - це текстовий файл, який визначає розклад завдань cron. Існує два типи файлів crontab.

Файли *crontab* користувачів іменуються у співвідношенні з іменем користувача, і їх розміщення залежить від операційної системи. Хоч можна

редагувати користувацькі файли *crontab* вручну, рекомендується використовувати команду crontab.

Команда *crontab* дозволяє встановити чи відкрити файл crontab для редагування. Можна використовувати команду crontab для перегляду, додавання, видалення чи зміни завдань cron, використовуючи наступні параметри:

1. *crontab -e* - відредагувати файл crontab чи створити його, якщо він ще не існує
2. *crontab -l* - показати вміст файла *crontab*
3. crontab -r - видалити поточний файл *crontab*
4. *crontab -i* - видалити поточний файл *crontab* із запитом перед видаленням
5. *crontab -u* - змінити інший файл *crontab*. Потрібні права системного адміністратора

Конфігураційний файл містить послідовність командних строк і розклад їх виклику. Пусті строки і строки, які починаються із символу “#” ігноруються. Інші строки є установками змінних оточення і командами cron.

Запис *crontab*-файла в загальному складається із семи полів:

хвилини години день місяць день_тижня ім’я_користувача команда

Допустимі значення часових параметрів:

Параметр	**Допустимий інтервал**
хвилини	0-59
години	0-23
день місяця	1-31
місяць	1-12
день тижня	0-7 (0-Нд, 1-Пн, 2-Вт, 3-Ср, 4-Чт, 5-Пт, 6-Сб, 7-Нд)

Поле може бути задано явно або шаблоном:

1. * - будь-яка цифра
2. ціле число
3. цілі числа через кому - завдання дискретної множини, наприклад 1, 2, 5
4. два цілих числа, розділених дефісом, відповідно до діапазону значень, наприклад 3-6

Приклад готової строки сценарія cron:

Виконати завдання о 18 годині 7 хвилин 13 травня якщо це п’ятниця
7 18 13 5 5 */home/www/myscript.pl*
Виконувати завдання раз в годину в 0 хвилин
0 */1 * * * */home/www/myscript.pl*
Виконувати завдання кожних сім годин в 0 хвилин
0 */7 * * * */home/www/myscript.pl*
Виконувати завдання по неділях в 10 годині 30 хвилин
30 10 * * 0 */home/www/myscript.pl*

2.3 Фреймворк.

2.3.1 Загальні поняття.

Фреймворки - це програмні продукти, які спрощують створення і підтримку технічно складних або загружених проєктів. Фреймворк, як правило, містить тільки базові програмні модулі, а всі специфічні для проєкта компоненти реалізуються розробником на їх основі. Тим самим досягається не тільки висока швидкість розробки, а й більша продуктивність і надійність рішень.

Однією з головних переваг у використанні фреймворків є те, що фреймворк визначає уніфіковану структуру для побудованих на його базі додатків. Тому додатки на фреймворках значно простіше супроводжувати і допрацьовувати, так як стандартизована структура організації компонентів зрозуміла всім розробникам на цій платформі і не потребує довгого розбору в архітектурі, щоб зрозуміти принцип роботи додатка чи знайти місце реалізації того чи іншого функціоналу. Більшість фреймворків для розробки веб-додатків використовує

парадигму *MVC* (модель-представлення-контроллер) - тобто дуже в багатьох фреймворках ідентичний підхід до організації компонентів додатка і це ще більше спрощує розуміння архітектури навіть на незнайомому розробнику фреймворку.

Проєктування архітектури ПЗ при розробці на фреймворку також дуже спрощується - в методологіях фреймворків зазвичай закладені кращі практики програмної інженерії і просто слідуючи цим правилам можна запобігти багатьом проблемам і помилкам в проєктуванні. По суті, фреймворк - це множина конкретних і абстрактних класів, зв’язаних між собою і впорядкованих згідно методології фреймворка. Конкретні класи зазвичай реалізують взаємні відношення між класами, а абстрактні представляють собою точки розширення, в яких, закладений у фреймворк базовий функціонал, може бути використаний “як є” або може бути адаптований під задачі конкретного додатка. Для забезпечення розширення можливостей в більшості фреймворків використовуються техніки об’єктно-орієнтованого програмування: наприклад, частини додатка можуть наслідуватись від базових класів фреймворка або окремі модулі можуть бути підключені як домішки.

Веб-фреймворки також багаті на готові реалізації багатьох функціональних можливостей. Розробникам при роботі над типовими задачами не потрібно “винаходити велосипед”, так як вони можуть використати вже створену спільнотою реалізацію. А це не тільки скорочує витрати часу і коштів, але й дозволяє добитись вищої стабільності рішення - компонент, який використовується і допрацьовується тисячами інших розробників зазвичай більш якісно реалізований і краще протестований на всіх можливих сценаріях, ніж рішення, яке може в адекватні терміни розробити один розробник чи навіть невелика команда.

2.3.2 *Yii2*.

Yii - це високоефективний, оснований на компонентній структурі *PHP* фреймворк для швидкої розробки веб-додатків. Він дозволяє максимально примінити концепцію повторного використання коду і може суттєво прискорити процес веб-розробки.

Yii можна використовувати для розробки будь-якого виду веб-додатків. Завдяки своїй основі компонентів, архітектурі і складній підтримці кешування, фреймворк підходить для розробки масштабних проєктів, таких як портали, форуми, системи управління контентом (*CMS*), систем електронної комерції і т. д.

Фреймворк *Yii* включає в себе наступні можливості:

- Реалізує для використання *MVC (Model-View-Controller)* архітектурний шаблон і сприяє організації кода на основі цього шаблона
- Дозволяє робити код простим і елегантним
- *Yii* є *full-stack* фреймворком, що надає велику кількість перевірених і готових до використання функцій: будівельник запитів і *ActiveRecord* для реляційних і *NoSQL* баз даних, *RESTful API*, підтримку багаторівневого кешування і т. д.
- *Yii* надзвичайно розширюваний фреймворк в якому можна замінити майже кожен кусочок кода і розробити потрібні розширення.
- Висока продуктивність є основною задачею *Yii*.

Yii не показник однієї людини, фреймворк підкріплений сильною командою розробників ядра, а також великою спільнотою професіоналів, які постійно сприяють його розвитку. Команда розробників продовжує уважно слідкувати за останніми тенденціями в області розвитку веб, а також правильними практиками розробки і особливостей в інших фреймворках і проєктах.

Найбільш значимі кращі практики і особливості, знайдені в інших місцях, регулярно включаються в ядро фреймворка і використовуються з допомогою простих і елегантних інтерфейсів.

В наш час існує дві основні версії фреймворка, які можна використовувати: 1.1 і 2.0. Версія 1.1. є старим поколінням і тепер знаходиться в режимі технічного обслуговування. Версія 2.0 представляє собою повністю переписаний *Yii*, з доданими найновішими технологіями і протоколами, в тому числі: *Composer*, *PSR*, простору імен, *Traits* і т. д.

Версія 2.0 представляє сучасне покоління фреймворка і буде отримувати основні зусилля в області розвитку в ході наступних кількох років. Для використання *Yii* 2.0 потребує *PHP* 5.4.0 або вище, а також базових знань ООП.

Висновок до розділу 2.

У другому розділі ми розглянули засоби, які потрібні для реалізації власної онлайнової системи управління чергою завдань. Так, було описано що таке сервер, які вони бувають і як працюють. Крім цього, що собою представляє ОС Лінукс, історія її виникнення, а також переваги використання.

Наступним пунктом другого розділу іде програмне забезпечення на сервері. Apache - це відкрите програмне забезпечення для створення веб-сервера. Його головна функція - швидка і надійна доставка контента в мережі Інтернет.

PHP - це мова програмування з відкритим вихідним кодом, над розвитком якого працюють програмісти-ентузіасти зі всього світу. Він має простий синтаксис, частково схожий на *Java* і *C++*. Це проєкт, який постійно розвивається. На даний момент актуальною є 8-ма версія мови. По статистиці, кожен шостий програмний продукт створений на *PHP*.

MySQL представляє собою одну з найрозповсюдженіших на сьогодні систем управління базами даних в мережі Інтернет. Дана система використовується для роботи з досить великими об’ємами інформації. Однак *MySQL* ідеально підходить як для невеликих, так і для масштабних інтернет-проєктів.

Cron - це демон планування, який виконує задачі з заданими інтервалами. Ці задачі називаються завданнями cron і в основному використовуються для автоматизації обслуговування чи адміністрування системи.

Останнім пунктом другого розділу був описаний фреймворк *Yii2*. Перед цим було дане загальне ознайомлення з терміном “фреймворк”. Фреймворки - це програмні продукти, які спрощують створення і підтримку технічно складних або загружених проєктів. Фреймворк, як правило, містить тільки базові програмні модулі, а всі специфічні для проєкта компоненти реалізуються розробником на їх основі. Тим самим досягається не тільки висока швидкість розробки, а й більша продуктивність і надійність рішень.

Yii - це високоефективний, оснований на компонентній структурі *PHP* фреймворк для швидкої розробки веб-додатків. Він дозволяє максимально примінити концепцію повторного використання коду і може суттєво прискорити процес веб-розробки.

У наступному розділі буде описано реалізацію власної онлайнової системи управління чергою завдань.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ОНЛАЙНОВОЇ СИСТЕМИ УПРАВЛІННЯ ЧЕРГОЮ ЗАВДАНЬ

3.1 Опис системи.

Для того щоб будь-яка інша система могла взаємодіяти з описуваною системою, реалізовано публічне *Api*, за допомогою якого створюються завдання. Всі завдання групуються за типами. Тип завдання містить інформацію кінцевого отримувача (*url* адреса отримувача), http метод відправки даних, пріоритет та кількість невдалих спроб передачі завдання на виконання. Кожне нове завдання створюється зі статусом "*Todo*".

Системний "Загарбник" постійно працює (.рисунок 3.1) і намагається взяти завдання в обробку. В кожній ітерації він намагається захопити невиконані завдання і передати їх на опрацювання "Виконавцю". Всі завдання на опрацювання йдуть в окремих потоках. За одну ітерацію він може захопити не більше ста завдань. Завдання беруться тільки по одному кожного типу і строго в порядку їх створення, що знаходяться в статусі "*Todo*" або "*Fail*". Завдання в статусі "*Fail*" можна взяти в роботу лише певну кількість разів, яка описана в типі цього завдання і не раніше однієї хвилини після останнього опрацювання. Якщо є завдання певного типу в одному із статусів ("*Capture*", "*In progress*", "*Error*"), то ця группа блокується і завдання з цієї групи більше в обробку не беруться. Коли завдання потрапляє в статус "*Error*", відправляється повідомлення на пошту адміністратору даної системи, який в свою чергу має вирішити, що робити з цим завданням щоб якнайшвидше розблокувати групу. Адміністратор може власноруч перевести завдання в статус "*Postponed*", що означає, що завдання відкладено для виконання на певний проміжок часу. Якщо "Виконавець" опрацював задачу, вона переводится в статус "*Done*".

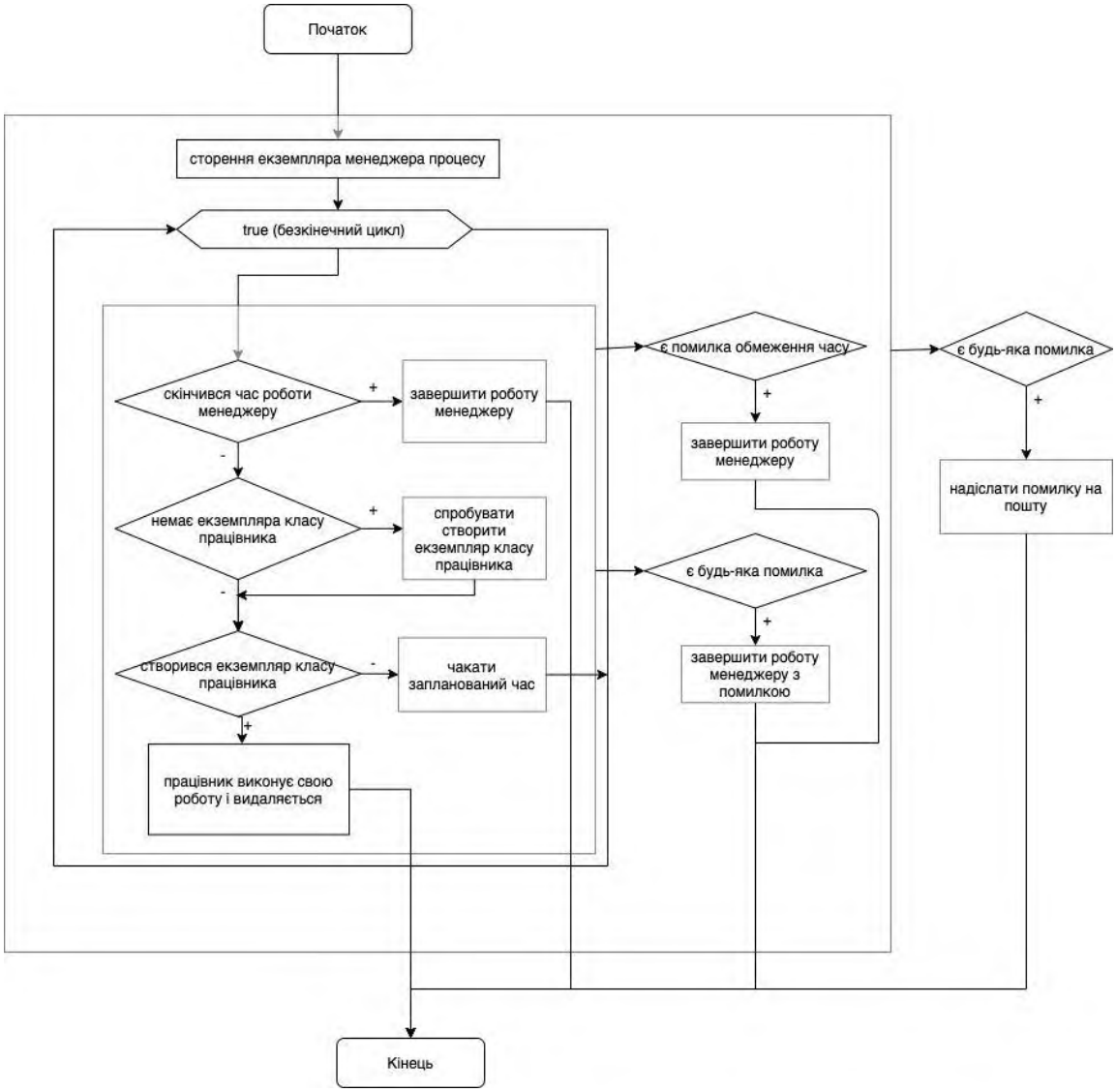

Рис. 3.1. Схема алгоритму процесу захоплення завдання.

Коли завдання надходить “Виконавцю” він переводить його в статус “*In progress*” та намагається відправити його кінцевому “Отримувачу”. В процесі передачі завдання всі запити до “Отримувача” логуються. Якщо завдання не отримується, сервіс тимчасово не працює, завдання переходить в статус “*Fail*”. Якщо “Виконавець” не зміг передати завдання в зв’язку з системними помилками завдання переходить в статус “*Error*”.

Для більш швидкої роботи системи, завдання та логи їх виконання, через одну хвилину після виконання переносяться в архів.

Всі системні помилки та повідомлення зберігаються в базі даних і відправляються на пошту адміністратору для більш швидкого реагування.

Життєвий цикл завдання:

1. Створення завдання. Завдання знаходиться в статусі “*Todo*”.

2. Процес “Загарбник” бере завдання в роботу. Завдання переходить в статус “*Capture*”. Завдання передається в процес “Виконавець” в новому потоці.

3. Процес “Виконавець” бере завдання в роботу. Завдання переходить в статус “*In progress*”. Завдання *http* запитом відправляється отримувачу:

a. Відправник відправляє завдання на виконання отримувачу.

b. Отримувач не зміг взяти завдання. Завдання переходить в статус “*Fail*”. В залежності від налаштувань через визначений час завдання знову буде доступне “Загарбнику”.

c. Записується лог спілкування “Відправника” з “Отримувачем”.

d. Відправник не зміг доставити завдання отримувачу. Завдання переходить в статус “*Error*”.

e. Виконавець не зміг коректно відпрацювати. Завдання переходить в статус “*Todo*”.

4. Виконане завдання переходить в архів.

3.2 База даних системи.

3.2.1 Опис таблиць.

Таблиця “*log*”. В ній записуються помилки системи.

```
CREATE TABLE `log` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `level` int(11) NOT NULL,
  `category` varchar(255) NOT NULL,
  `log_time` float NOT NULL,
  `prefix` text NOT NULL,
  `message` text NOT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_log_level` (`level`),
  KEY `idx_log_category` (`category`)
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

level - (1 - *Error*, 2 - *Warning*, 4 - *Info*) -

Error - повідомлення про помилку - це повідомлення, яке вказує на ненормальне завершення роботи додатка і може потребувати обробки розробника.

Warning - попереджуюче повідомлення - це повідомлення, що вказує, що виникла якась ненормальна ситуація, але додаток може продовжувати працювати.

Info - інформаційне повідомлення - це повідомлення, яке містить деяку інформацію для розробників.

category - абстрактна назва категорії, яка показує в якій частині коду записано це повідомлення;

log_time - час, коли було записано це повідомлення (*timestamp* формат);

prefix - допоміжні дані про повідомлення в форматі: *[userIP][userID][sessionID];*

message - текст повідомлення.

Таблиця "*apiLog*". Вона зберігає інформацію про зовнішні запити до системи.

```
CREATE TABLE `apiLog` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `url` varchar(255) NOT NULL,
  `method` varchar(255) NOT NULL,
  `authCredentials` varchar(255) NOT NULL,
  `userIp` varchar(255) NOT NULL,
  `headers` text NOT NULL,
  `rawBody` text NOT NULL,
  `responseStatus` int(11) unsigned NOT NULL,
  `createTime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=64 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

url - адреса запиту;

method - *http* метод запиту (*get*, *post*, *put*);

authCredentials - ключ доступа для користувача (["*user login*", "*user password*"]);

userIp - *Ip* адреса користувача;

headers - *http* заголовки запиту;

rawBody - тіло запиту;

responseStatus - *http* код відповіді системи;

createTime - час створення запису.

Таблиця “*workLog*”. Вона зберігає інформацію про виконання *cron* процесів. Необхідна для контролю процесів щоб процеси кожного типу виконувались суворо описаними налаштуваннями.

```
CREATE TABLE `workLog` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `typeId` int(11) unsigned NOT NULL,
  `statusId` tinyint(3) unsigned NOT NULL,
  `startTime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `finishTime` timestamp NULL DEFAULT NULL,
  `cnt` int(11) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  KEY `typeId` (`typeId`),
  KEY `statusId` (`statusId`),
  CONSTRAINT `FK-Status` FOREIGN KEY (`statusId`) REFERENCES `workLogStatus` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `FK-Type` FOREIGN KEY (`typeId`) REFERENCES `workType` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=8740 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

typeId - порядковий номер типу процесу, зв’язок з таблицею “*workType*”;

statusId - порядковий номер статусу процесу, зв’язок з таблицею “*workLogStatus*”;

startTime - час початку виконання процесу;

finishTime - час закінчення виконання процесу;

cnt - кількість виконаних завдань.

Таблиця “*workLogStatus*”. Вона зберігає список статусів, в яких може знаходитись cron завдання. *In progress* - завдання в процесі виконання. *Done* - завдання завершено. *Failed* - завдання завершилось з помилкою. *Killed* - процес примусово завершений.

```
CREATE TABLE `workLogStatus` (
  `id` tinyint(3) unsigned NOT NULL,
  `name` varchar(64) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

name - назва статусу.

Таблиця "*workType*". Вона зберігає налаштування для типів завдань.

Існуючі типи:

1. *TaskCapture* - процес захвату завдання в роботу. Час роботи одна година, максимальна кількість потоків - один, час очікування появи завдання - одна секунда.

2. *TaskImplementation* - процес, який відповідає за виконання завдання. Час роботи необмежений, максимальна кількість потоків - сто, час очікування появи завдання - одна секунда.

3. *TaskArchiving* - процес для архівування виконаних завдань. Час роботи одна година, максимальна кількість потоків - один, час очікування появи завдання - одна секунда.

```
CREATE TABLE `workType` (
  `id` int(11) unsigned NOT NULL,
  `name` varchar(128) NOT NULL,
  `maxTime` int(11) unsigned NOT NULL,
  `maxStream` tinyint(3) NOT NULL DEFAULT '1',
  `waitTime` int(11) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

name - назва типу;

maxTime - максимальний час виконання;

maxStream - максимальна кількість потоків із завданнями відповідного типу;

waitTime - час очікування.

Таблиця "*task*". Вона зберігає інформацію про завдання.

```
CREATE TABLE `task` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `typeId` int(11) unsigned NOT NULL,
  `statusId` tinyint(3) unsigned NOT NULL,
  `workerId` int(11) unsigned DEFAULT NULL,
  `data` text,
  `createTime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `updateTime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `finishTime` timestamp NULL DEFAULT NULL,
  `waitToTime` timestamp NULL DEFAULT NULL,
  `cntUnsuccessfulAttempts` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  KEY `typeId` (`typeId`),
  KEY `statusId` (`statusId`),
  KEY `workerId` (`workerId`),
  CONSTRAINT `FK-TaskStatus` FOREIGN KEY (`statusId`) REFERENCES `taskStatus` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `FK-TaskType` FOREIGN KEY (`typeId`) REFERENCES `taskType` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=28187 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

typeId - порядковий номер типу завдання, зв'язок з таблицею "*taskType*";

statusId - порядковий номер статусу завдання, зв'язок з таблицею "*taskStatus*";

workerId - порядковий номер процесу, який виконує це завдання, зв'язок з таблицею "*workLog*";

data - тіло завдання;

createTime - час створення завдання;

updateTime - час оновлення завдання;

finishTime - час закінчення виконання завдання;

waitToTime - час, після якого завдання знову може бути взяте в роботу, після невдалого виконання;

cntUnsuccessfulAttempts - кількість невдалих спроб виконати завдання.

Таблиця "*taskArchive*". Вона зберігає інформацію про вже виконані завдання.

```
CREATE TABLE `taskArchive` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `taskId` int(11) unsigned NOT NULL,
  `typeId` int(11) unsigned NOT NULL,
  `statusId` tinyint(3) unsigned NOT NULL,
  `workerId` int(11) unsigned DEFAULT NULL,
  `data` text,
  `createTime` timestamp NOT NULL,
  `updateTime` timestamp NOT NULL,
  `finishTime` timestamp NULL DEFAULT NULL,
  `waitToTime` timestamp NULL DEFAULT NULL,
  `cntUnsuccessfulAttempts` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=8637 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

taskId - порядковий номер завдання, зв'язок з таблицею "*task*"

typeId - порядковий номер типу завдання, зв'язок з таблицею "*taskType*";

statusId - порядковий номер статусу завдання, зв'язок з таблицею "*taskStatus*";

workerId - порядковий номер процесу, який виконує це завдання, зв'язок з таблицею "*workLog*";

data - тіло завдання;

createTime - час створення завдання;

updateTime - час оновлення завдання;

finishTime - час закінчення виконання завдання;

waitToTime - час, після якого завдання знову може бути взяте в роботу, після невдалого виконання;

cntUnsuccessfulAttempts - кількість невдалих спроб виконати завдання.

Таблиця "*taskImplementation*". Вона зберігає інформацію про процес виконання завдання.

```
CREATE TABLE `taskImplementation` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `taskId` int(11) unsigned NOT NULL,
  `rawRequest` mediumtext NOT NULL,
  `rawResponse` mediumtext,
  `createTime` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `workTime` double(7,4) NOT NULL DEFAULT '0.0000',
  PRIMARY KEY (`id`),
  KEY `taskId` (`taskId`),
  CONSTRAINT `FK-Task` FOREIGN KEY (`taskId`) REFERENCES `task` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=8640 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

taskId - порядковий номер завдання, зв'язок з таблицею "*task*";

rawRequest - *http* дані запиту (*http* заголовки та тіло запиту) до отримувача;

rawResponse - *http* дані відповіді отримувача на отримання завдання;

createTime - час створення запису;

workTime - час виконання запиту.

Таблиця "*taskImplementationArchive*". Вона зберігає дані про виконані завдання.

```
CREATE TABLE `taskImplementationArchive` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `taskImplementationId` int(11) unsigned NOT NULL,
  `taskId` int(11) unsigned NOT NULL,
  `rawRequest` mediumtext NOT NULL,
  `rawResponse` mediumtext,
  `createTime` timestamp NOT NULL,
  `workTime` double(7,4) NOT NULL DEFAULT '0.0000',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=8616 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

taskImplementationId - порядковий номер запису спілкування з отримувачем завдання, зв'язок з таблицею "*taskImplementation*";

taskId - порядковий номер завдання, зв'язок з таблицею "*task*";

rawRequest - *http* дані запиту (*http* заголовки та тіло запиту) до отримувача;

rawResponse - *http* дані відповіді отримувача на отримання завдання;

createTime - час створення запису;

workTime - час виконання запиту.

Таблиця "*taskStatus*". Вона зберігає список можливих статусів завдання.

```
CREATE TABLE `taskStatus` (
  `id` tinyint(3) unsigned NOT NULL AUTO_INCREMENT,
```

```
  `name` varchar(56) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

name - назва статусу;

Таблиця "*taskType*". Вона зберігає типи завдань, а також інформацію про отримувача, *url* адресу отримувача, http метод відправки запиту, приорітет і максимальну кількість можливих невдалих спроб отримання завдання.

```
CREATE TABLE `taskType` (
  `id` int(11) unsigned NOT NULL,
  `path` varchar(255) NOT NULL,
  `method` enum('GET','POST') NOT NULL DEFAULT 'POST',
  `name` varchar(255) NOT NULL,
  `priority` tinyint(3) unsigned NOT NULL,
  `cntAttempts` tinyint(3) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  KEY `priority_cntAttempts` (`priority`,`cntAttempts`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

id - порядковий номер запису;

path - *url* адреса отримувача завдання відповідного типу;

method - *http* метод запиту (може бути тільки *get* або *post*);

name - назва типу завдання;

priority - пріоритет виконання завдання;

cntAttempts - максимальна кількість невдалих спроб.

3.2.2 Зв’язок між таблицями.

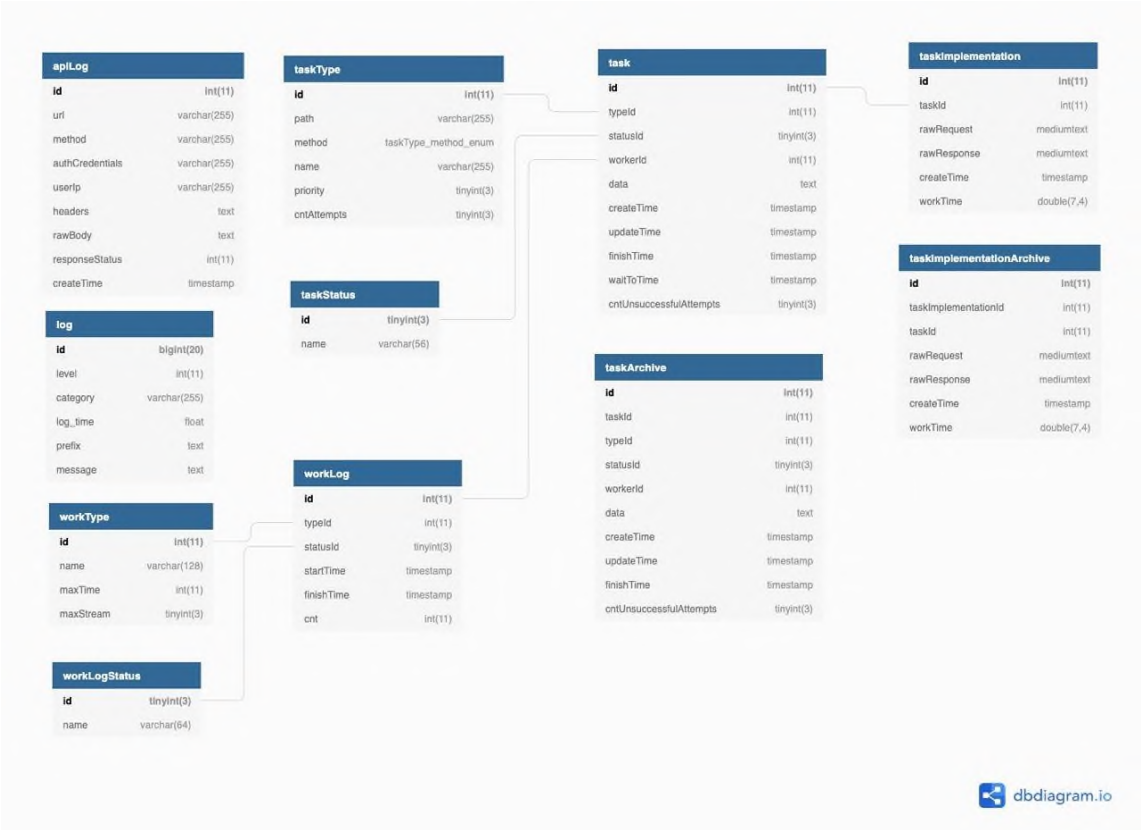

Рис. 3.2. Зв’язок між таблицями

3.3 Опис модулів програми.

3.3.1 Restful Api.

Для додавання завдань і типів завдань реалізовано публічне *Api*, через яке зовнішній сервіс може спілкуватися з розробленою онлайновою системою управління чергою завдань. Вхідні і вихідні дані у форматі *Json*. В системі реалізовано два контроллера: один для роботи з типами завдань, а другий для роботи з самими завданнями.

Task *controller*, методи:

1. Отримання списка всіх завдань

a. *Uri: /tasks*;

b. *Http* метод: *get*;

c. Приклад: *curl --user userName: -i "http://tq.local/tasks"*;

2. Отримання інформації по конкретному завданню

. *Uri: /tasks/<id>*, де *id* - це порядковий номер завдання;

a. *Http* метод: *get*;

b. Приклад: *url --user userName: -i -X GET "http://tq.local/tasks/123"*;

3. Створення нового завдання

. *Uri: /tasks*;

a. *Http* метод: *post*;

b. Приклад: *curl --user userName: -i -H "Content-Type: application/json" -X POST "http://tq.local/tasks" -d '{"typeId":1,"statusId":1}'*;

4. Редагування завдання

. *Uri: /tasks/<id>*, де *id* - це порядковий номер завдання;

a. *Http* метод: *put*;

b. Приклад: *curl --user userName: -i -H "Content-Type: application/json" -X PUT "http://tq.local/tasks/30" -d '{"typeId":100}'*;

Приклад структури завдання:

```
{
    "typeId": 1,
    "statusId": 7,
    "data": "{\"data\":2}",
    "createTime": "2021-01-23 18:57:22",
    "updateTime": "2021-01-23 20:15:04",
    "finishTime": null,
    "waitToTime": "2021-01-23 20:12:28",
    "cntUnsuccessfulAttempts": 4
}
```

Формат даних створення завдання:

1. Обов’язкові поля: *typeId*, *statusId*;
2. Необов’язкові поля: *data*, *cntUnsuccessfulAttempts*;
3. *typeId*, *statusId*, *cntUnsuccessfulAttempts* - тип даних *integer*;
4. *data* - тип даних *string*;

Task type controller, методи:

1. Отримання списка всіх типів завдань

a. *Uri: /tasks-type*;

b. *Http* метод: *get*;

c. Приклад: *curl --user userName: -i "http://tq.local/tasks-type"*;

2. Отримання інформації по конкретному типу завдання

. *Uri: /tasks-type/<id>*, де *id* - це порядковий номер завдання;

a. *Http* метод: *get*;

b. Приклад: *url --user userName: -i -X GET "http://tq.local/tasks-type/123"*;

3. Створення нового типу завдання

. *Uri: /tasks-type*;

a. *Http* метод: *post*;

b. Приклад: *curl --user userName: -i -H "Content-Type: application/json" -X POST "http://tq.local/tasks-type" -d '{"path":"url","name":"tasktypename","priority":1,"cntAttempts":5}'*;

4. Редагування типу завдання

. *Uri: /tasks-type/<id>*, де *id* - це порядковий номер завдання;

a. *Http* метод: *put*;

b. Приклад: *curl --user userName: -i -H "Content-Type: application/json" -X PUT "http://tq.local/tasks-type/30" -d '{"priority":10}'*;

Приклад структури типу завдання:

```
{
    "id": 1,
    "path": "http://domain/uri",
    "method": "POST",
    "name": "name",
    "priority": 6,
    "cntAttempts": 2,
  }
```

Формат даних створення типу завдання:

1. Обов’язкові поля: *path, name, priority, cntAttempts*;
2. Необов’язкові поля: *method*;
3. *priority, cntAttempts* - тип даних *integer*;

4. *path, method, name* - тип даних *string*;

3.3.2 Cron процеси.

Task Capture.

Потрібен для того щоб взяти завдання, яке перше в черзі по пріоритету. З кожного типу завдань можна взяти тільки одну. Для коректної роботи процесу використовується компонент Cron Manager, який слідкує за тим, щоб процеси виконувались строго обмежений час і визначає чи можуть декілька процесів одного типу працювати одночасно. Task Capture запускається кожну годину і працює одну годину. На протязі цього часу він, за допомогою помічника worker, перевіряє наявність існуючих завдань щоб взяти в роботу. Якщо завдань немає, чекає визначену кількість часу (це значення береться з таблиці *workType*, поле *waitTime*) і знову перевіряє. Якщо завдання знайдено worker його обробляє і процес починається з початку.

Task Implementation.

Це процес, який обробляє завдання. Кількість паралельних потоків і час роботи визначається в таблиці *workType*. Процес на вхід отримує get параметром *id* завдання, що потрібно виконати. В процесі його роботи також створюється менеджер, який слідкує за потоками і часом. З допомогою worker виконується завдання.

Archiving.

Процес, який архівує виконані завдання, для звільнення таблиці *task*, що прискорює роботу системи. Кількість паралельних потоків і час роботи визначається в таблиці *workType*. В процесі його роботи також створюється менеджер, який слідкує за потоками і часом. В процесі роботи, за допомогою *worker*, архівує виконані завдання. Архівуються лише завдання, після виконання яких пройшла одна хвилина.

Cleaner.

Запускається кожну хвилину і слідкує за всіма процесами. Якщо якийсь процес виконується більше відведеного для нього часу, він його примусово завершує.

Висновок до розділу 3.

В даному розділі було описано основні моменти реалізації онлайнової системи управління чергою завдань. Наведено опис таблиць, де детально розглянута кожна таблиця, з назвами полів та для чого вона потрібна, а також графічне зображення зв’язків між таблицями. В підрозділі про модулі програми було описано *Restful Api* та *Cron* процеси.

ВИСНОВКИ

Під час виконання дипломного проєкту було досліджено поняття черги та її необхідності. Було пояснено та обґрунтовано актуальність онлайнових систем управління чергою завдань в сучасному світі, з чого можна зробити висновок, що такі системи мають ряд вагомих переваг використання: економія ресурсів, надійність та гарантія почергової обробки.

Як приклад було розглянуто існуючі системи, а перед цим описано яких типів вони бувають: “Точка-точка”, “Видавець-підписник”, “Гібридна модель”.

В класичних системах обміну повідомленнями модель “точка-точка” реалізується через черги. Черга діє як буфер *FIFO* (перший зайшов, перший вийшов), на який може підписатися один або декілька споживачів. Кожне повідомлення доставляється тільки одному з підписаних споживачів. Черги зазвичай стараються справедливо розділити повідомлення між споживачами. Тільки один споживач отримає дане повідомлення.

В класичних системах обміну повідомленнями модель обміну повідомленнями “публікація-підписка” реалізується через топіки. Топік представляє такий самий спосіб широкомовлення, як механізм конференц-зв’язку. Коли повідомлення відправляється в топік, воно розподіляється по всім підписаним користувачам. Спікера не хвилює скільки людей в даний момент приєднались до дзвінка - система гарантує, що будь-хто, хто підключиться в даний момент почує про що говориться.

У випадках коли потрібен адресат (загальний термін для черг і топіків), який розподіляє повідомлення в основному як топік, так, що кожне повідомлення відправляється в окрему систему, зацікавлену в цих повідомленнях, але і також в якій кожна система може визначити декілька споживачів, які отримують вхідні повідомлення, що більше схоже на чергу. Тип зчитування в цьому випадку - один раз для кожної зацікавленої сторони. Ці гібридні адресати часто потребують надійності, так що якщо споживач відключається, повідомлення, які відправляються в цей час, приймаються після повторного підключення споживачів.

Щоб створити свою систему для управління чергою завдань було проведено дослідження *Active MQ* та *Kafka* і коротко описано основні моменти в їх роботі: збереження та відправка повідомлень. Дані дві системи відносяться до типів “Видавець-Підписник” та Гібридна модель і є абсолютно різними.

Active MQ працює як поштове відділення: всі отримані повідомлення записуються в журнал і очікують, коли споживач буде готовий їх прийняти. Тільки після того як система отримає підтвердження, що повідомлення доставлено, воно може видалитись з журналу.

Kafka об’єднала обмін повідомленнями типу “публікація-підписка” і “точка-точка” в рамках одного виду адресата - топіка. В кожного топіка *Kafka* є свій журнал. Продюсери, що відправляють повідомлення в *Kafka*, дописують в цей журнал, а отримувачі читають з журнала з допомогою покажчиків, які постійно переміщуються вперед. Періодично *Kafka* видаляє найстаріші частини журнала, незалежно від того, чи були повідомлення в цих частинах прочитані. В *Kafka* брокер не піклується про те, прочитані повідомлення чи ні - це відповідальність клієнта.

Свою систему управління чергою завдань було вирішено розробляти по типу “Точка-точка”, що зробить її простішою але при цьому досить ефективною.

Для реалізації власної онлайнової системи управління чергою завдань було використано популярну мову програмування *PHP* та фреймворк *Yii2*, СУБД *MySQL* та планувальник завдань *Cron*.

Життєвий цикл завдання у реалізованій системі управління чергою завдань:

1. Створення завдання. Завдання знаходиться в статусі “*Todo*”.
2. Процес “Загарбник” бере завдання в роботу. Завдання переходить в статус “*Capture*”. Завдання передається в процес “Виконавець” в новому потоці.
3. Процес “Виконавець” бере завдання в роботу. Завдання переходить в статус “*In progress*”. Завдання *http* запитом відправляється отримувачу.
4. Виконане завдання переходить в архів.

БД системи містить такі таблиці:

1. Таблиця “*log*”. В ній записуються помилки системи.

2. Таблиця “*apiLog*”. Вона зберігає інформацію про зовнішні запити до системи.

3. Таблиця “*workLog*”. Вона зберігає інформацію про виконання *cron* процесів. Необхідна для контролю процесів щоб процеси кожного типу виконувались суворо описаними налаштуваннями.

4. Таблиця “*workLogStatus*”. Вона зберігає список статусів, в яких може знаходитись cron завдання. *In progress* - завдання в процесі виконання. *Done* - завдання завершено. *Failed* - завдання завершилось з помилкою. *Killed* - процес примусово завершений.

5. Таблиця “*workType*”. Вона зберігає налаштування для типів завдань.

6. Таблиця “*task*”. Вона зберігає інформацію про завдання.

7. Таблиця “*taskArchive*”. Вона зберігає інформацію про вже виконані завдання.

8. Таблиця “*taskImplementation*”. Вона зберігає інформацію про процес виконання завдання.

9. Таблиця “*taskImplementationArchive*”. Вона зберігає дані про виконані завдання.

10. Таблиця “*taskStatus*”. Вона зберігає список можливих статусів завдання.

11. Таблиця “*taskType*”. Вона зберігає типи завдань, а також інформацію про отримувача, *url* адресу отримувача, *http* метод відправки запиту, приорітет і максимальну кількість можливих невдалих спроб отримання завдання.

Для додавання завдань і типів завдань реалізовано публічне *Api,* через яке зовнішній сервіс може спілкуватися з розробленою онлайновою системою управління чергою завдань. Для того щоб взяти завдання в роботу, обробити його, заархівувати чи примусово завершити використовуються *Cron*-процеси.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 3008-95. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. – 39 с.
2. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.
3. Поняття черги - *https://aws.amazon.com/ru/message-queue/*
4. Брокери повідомлень - *https://habr.com/ru/post/466385/*
5. Сервер - *https://www.seonews.ru/glossary/server/*
6. *ОС Linux* - *http://mirubuntu.ru/chto-takoe-linux/*
7. *PHP* - *http://www.php.su/php/?php*
8. *Apache* - *https://eternalhost.net/blog/hosting/web-server-apache*
9. *Yii2* - *https://www.yiiframework.com/doc/guide/1.1/ru/quickstart.what-is-yii*
10. *MySQL* - *https://ru.hostings.info/schools/bazy-dannyh.html*
11. *Cron* - *http://www.codenet.ru/webmast/php/cron.php*

Додаток А

Лістинг коду основного програмного модулю

commands/AbstractCronController.php

```
<?php

namespace app\commands;

use app\components\log\CronManager;
use app\components\log\CronManagerFactory;
use app\components\log\ManagerCreationException;
use app\components\log\WaitException;
use yii\console\Controller;

/**
* Class AbstractCronController
* @package app\commands
*/
class AbstractCronController extends Controller
{
  protected static int $maxTimeWaitingManager = 3600;

  /**
   * @param int $workTypeId
   * @return CronManager
   * @throws ManagerCreationException
   */
  protected function getCronManager(int $workTypeId): CronManager
  {
     $startTime = time();
     do {
        try {
```

```
            $manager = CronManagerFactory::getManager($workTypeId);
        } catch (WaitException $e) {
            // do nothing
            // wait until the thread is released
            sleep(1);
            continue;
        }
        if (!is_null($manager)) {
            return $manager;
        }
    } while ((time() - $startTime) <= static::$maxTimeWaitingManager);

    throw new ManagerCreationException("Didn't get a manager.");
  }
}
```

commands/TaskCaptureController.php

```
<?php

namespace app\commands;

use app\components\log\DeadException;
use app\components\taskqueue\workers\WorkerFactory;
use app\models\worklog\WorkType;
use yii\console\ExitCode;
use Yii;
use Throwable;

/**
* Class TaskCaptureController
* @package app\commands
```

```
*/
class TaskCaptureController extends AbstractCronController
{

  /**
   * @return int
   */
  public function actionCapture()
  {
     try {
     $manager = $this->getCronManager(WorkType::TYPE_TASK_CAPTURE);
        while (true) {
           try {

              if ($manager->isExpired()) {
                 // waiting for the next launch
                 $manager->done();
                 exit;
              }

              if (!isset($worker)) {
                 $worker = WorkerFactory::getTaskCapture($manager->getWorkId());
              }
              if (is_null($worker)) {
                 // if there are no tasks, we will wait and try to capture again
                 sleep($manager->getWaitingTime());
                 continue;
              }

              // to perform the task
```

```
                $worker->process();
                $manager->updateIteration();
                unset($worker);
            } catch (DeadException $e) {
                $manager->done();
                Yii::warning($e->getMessage(), static::class);
                exit;
            } catch (Throwable $e) {
                $manager->fail();
                Yii::error($e->getMessage(), static::class);
            }
        }
    } catch (Throwable $e) {
        Yii::error($e->getMessage(), static::class);
    }

    return ExitCode::OK;
  }
}
```

commands/TaskImplementationController.php

```
<?php

namespace app\commands;

use app\components\log\CronManager;
use app\components\log\DeadException;
use app\components\taskqueue\workers\WorkerFactory;
use app\models\tasks\Task;
use app\models\tasks\TaskStatus;
use app\models\worklog\WorkType;
```

```
use yii\console\ExitCode;
use Throwable;
use Yii;

/**
* Class TaskImplementationController
* @package app\commands
*/
class TaskImplementationController extends AbstractCronController
{
  /**
   * @param $taskId
   * @return int
   */
  public function actionImplementation($taskId)
  {
    if (!$task = Task::findOne($taskId)) {
      Yii::error("Unknown task. [taskId=$taskId]", static::class);
      return ExitCode::SOFTWARE;
    }

    $manager = null;
    try {
      $manager = $this->getCronManager(WorkType::TYPE_TASK_IMPLEMENTATION);
    } catch (Throwable $e) {
      Yii::error($e->getMessage(), static::class);
    }

    try {
      if (!$manager) {
        Yii::error('Can not create cron manager', static::class);
```

```
                $this->resetTask($task);
                return ExitCode::UNSPECIFIED_ERROR;
            }
        } catch (Throwable $e) {
            Yii::error($e->getMessage(), static::class);
            return ExitCode::UNSPECIFIED_ERROR;
        }

        try {
            if (!$worker = WorkerFactory::getTaskImplementation($manager->getWorkId(), $taskId)) {
                Yii::error('Can not create worker', static::class);
                $this->errorTask($task);
                return ExitCode::UNSPECIFIED_ERROR;
            }

            $worker->process();
            $manager->updateIteration();
            $manager->done();

            return ExitCode::OK;
        } catch (DeadException $e) {
            $manager->done();
            Yii::warning($e->getMessage(), static::class);
            exit;
        } catch (Throwable $e) {
            Yii::error($e->getMessage(), static::class);
            $this->processFail($manager, $task);
        }

        return ExitCode::UNSPECIFIED_ERROR;
    }
```

```
/**
 * @param Throwable $e
 * @param CronManager $manager
 * @param Task $task
 */
private function processFail(CronManager $manager, Task $task)
{
    try {
        $manager->fail();
        $this->errorTask($task);
    } catch (Throwable $e) {
        Yii::error($e->getMessage(), static::class);
    }
}

/**
 * @param Task $task
 * @throws Throwable
 * @throws \yii\db\StaleObjectException
 */
private function resetTask(Task $task)
{
    $task->setStatus(TaskStatus::STATUS_TODO);
}

/**
 * @param Task $task
 * @throws Throwable
 * @throws \yii\db\StaleObjectException
 */
private function errorTask(Task $task)
```

```
    {
        $task->processError();
    }
}
```

commands/ArchivingController.php

```
<?php

namespace app\commands;

use app\components\log\DeadException;
use app\components\log\ManagerCreationException;
use app\components\taskqueue\workers\WorkerFactory;
use app\models\worklog\WorkType;
use yii\console\ExitCode;
use Yii;
use Throwable;

/**
* Class ArchivingController
* @package app\commands
*/
class ArchivingController extends AbstractCronController
{

    /**
     * @return int
     */
    public function actionIndex()
    {
        try {
```

```
            $manager = $this->getCronManager(WorkType::TYPE_TASK_ARCHIVING);
        } catch (ManagerCreationException $e) {
            $manager = null;
        }

        if (!$manager) {
            Yii::error('Can not create cron manager', static::class);
            return ExitCode::UNSPECIFIED_ERROR;
        }

        try {
            while (true) {
                if ($manager->isExpired()) {
                    // waiting for the next launch
                    $manager->done();
                    return ExitCode::OK;
                }

                if (!isset($worker)) {
                    $worker = WorkerFactory::getTaskArchiving();
                }
                if (is_null($worker)) {
                    // if there are no tasks, we will wait and try to archiving again
                    sleep($manager->getWaitingTime());
                    continue;
                }

                $worker->process();
                $manager->updateIteration();
                unset($worker);
            }
```

```
        } catch (DeadException $e) {
            $manager->done();
            Yii::warning($e->getMessage(), static::class);
            exit;
        } catch (Throwable $e) {
            Yii::error($e->getMessage(), static::class);
            $manager->fail();
        }

        return ExitCode::UNSPECIFIED_ERROR;
    }
}
```

commands/CleanerController.php

```
<?php

namespace app\commands;

use app\components\log\CronCleaner;
use yii\console\Controller;

/**
* Class CleanerController
* @package app\commands
*/
class CleanerController extends Controller
{
    /**
     * @throws \yii\db\Exception
     */
    public function actionIndex()
```

```
    {
        CronCleaner::kill();
    }
}

components/log/CronCleaner.php
<?php

namespace app\components\log;

use app\models\worklog\LogStatus;
use Yii;

/**
* Class CronCleaner
* @package app\components\log
*/
class CronCleaner
{
    /**
     * Clear hung tasks
     * @throws \yii\db\Exception
     */
    public static function kill()
    {
        $query = <<<SQL
UPDATE workLog set statusId = :killed where statusId = :progress AND id IN (
    select * from (
        select WL.id from workLog as WL join workType as WT on
WL.typeId=WT.id
```

```
        where WL.statusId = :progress AND WL.startTime < NOW() - INTERVAL
WT.maxTime SECOND
      ) as tmp
    );
    SQL;
        Yii::$app->db->createCommand(
          $query,
          [
            ':killed'=> LogStatus::KILLED,
            ':progress'=> LogStatus::IN_PROGRESS,
          ]
        )->execute();
      }
    }

    components/log/CronManager.php
    <?php

    namespace app\components\log;

    use yii\db\Expression;
    use app\models\worklog\LogStatus;
    use app\models\worklog\Log;
    use Exception;

    /**
    * Class CronManager
    * @package app\components\log
    */
    class CronManager
    {
```

```
    private Log $log;

    /**
     * CronManager constructor.
     * @param Log $log
     * @throws Exception
     */
    public function __construct(Log $log)
    {
        if ($log->statusId != LogStatus::IN_PROGRESS) {
            throw new Exception("Incorrect work log status. [LogId={$log->id}]");
        }

        $this->log = $log;

        if (!$log->workType) {
            throw new Exception('Incorrect work type');
        }
    }

    /**
     * @throws DeadException
     * @throws \Throwable
     */
    public function updateIteration()
    {
        $this->log->cnt = new Expression('cnt + 1');

        if (!$this->log->update()) {
            throw new Exception("The manager could not update the working
process. [LogId={$this->log->id}]");
        }
```

```
            if($this->isKilled()) {
                throw new DeadException();
            }
        }

        /**
         * @throws \Throwable
         */
        public function done()
        {
            $this->log->finishTime = new Expression('NOW()');
            $this->log->statusId = LogStatus::DONE;

            if (!$this->log->update()) {
                throw new Exception("The manager could not done the working process.
[LogId={$this->log->id}]");
            }
        }

        /**
         * @throws \Throwable
         * @throws \yii\db\StaleObjectException
         */
        public function fail()
        {
            if ($this->log->statusId != LogStatus::IN_PROGRESS) {
                \Yii::warning("Status cannot be changed to fail. [LogId={$this->log-
>id}]");
            }

            $this->log->finishTime = new Expression('NOW()');
```

```
            $this->log->statusId = LogStatus::FAILED;
            if (!$this->log->update()) {
                throw new Exception("The manager could not fail the working process.
[LogId={$this->log->id}]");
            }
        }

        /**
         * @return bool
         */
        public function isExpired(): bool
        {
            if (!$this->log->workType->maxTime) {
                return false;
            }

            return time() - strtotime($this->log->startTime) >= $this->log->workType-
>maxTime;
        }

        /**
         * @return int
         */
        public function getWaitingTime(): int
        {
            return (int)$this->log->workType->waitTime;
        }

        /**
         * @return int
         */
        public function getWorkId(): int
```

```
    {
        return (int)$this->log->id;
    }

    /**
     * @return bool
     */
    private function isKilled(): bool
    {
        $this->log->refresh();

        return $this->log->statusId == LogStatus::KILLED;
    }
}
```

components/log/CronManagerFactory.php

```
<?php

namespace app\components\log;

use app\models\worklog\Log;
use app\models\worklog\WorkType;
use Exception;

/**
* Class CronManagerFactory
* @package app\components\log
*/
class CronManagerFactory
{
    /**
```

```
     * @param int $workTypeId
     * @return CronManager
     * @throws WaitException
     * @throws Exception
     */
    public static function getManager(int $workTypeId)
    {
        $workType = WorkType::findOne($workTypeId);
        if (!$workType) {
            throw new Exception("Unknown work type");
        }

        $log = Log::createInstance($workType);
        if (!$log) {
            throw new WaitException();
        }

        return new CronManager($log);
    }
}
```

components/log/DeadException.php

```
<?php

namespace app\components\log;

use Exception;
use Throwable;

class DeadException extends Exception
{
```

```
    public function __construct($message = "The process was killed", $code = 0,
Throwable $previous = null)
    {
        parent::__construct($message, $code, $previous);
    }
}
```

components/log/ManagerCreationException.php

```
<?php

namespace app\components\log;

/**
* Class ManagerCreationException
* @package app\components\log
*/
class ManagerCreationException extends \Exception
{

}
```

components/log/WaitException.php

```
<?php

namespace app\components\log;

use Exception;
use Throwable;

/**
* Class WaitException
```

```
 * @package app\components\log
 */
class WaitException extends Exception
{
  /**
   * WaitException constructor.
   * @param string $message
   * @param int $code
   * @param Throwable|null $previous
   */
  public function __construct($message = "Waiting task", $code = 0, Throwable $previous = null)
  {
    parent::__construct($message, $code, $previous);
  }
}
```

components/taskqueue/jobs/JobInterface.php

```
<?php

namespace app\components\taskqueue\jobs;

/**
* Interface JobInterface
* @package app\components\taskqueue\jobs
*/
interface JobInterface
{
  /**
   * @return bool
   */
```

```
    public function process(): bool;
}

components/taskqueue/jobs/TaskArchiving.php
<?php

namespace app\components\taskqueue\jobs;

use app\models\tasks\Task;

/**
* Class TaskArchiving
* @package app\components\taskqueue\jobs
*/
class TaskArchiving implements JobInterface
{
    // In such a number of seconds after processing the task, it goes into the archive
    const ARCHIVING_TIME = 60;

    private array $tasks;

    /**
     * TaskArchiving constructor.
     * @param Task[] $tasks
     */
    public function __construct(array $tasks)
    {
        $this->tasks = $tasks;
    }

    /**
```

```
     * @return bool
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    public function process(): bool
    {
        foreach ($this->tasks as $task) {
            $task->moveToArchive();
        }

        return true;
    }

}
```

components/taskqueue/jobs/TaskCapture.php

```
<?php

namespace app\components\taskqueue\jobs;

/**
* Class TaskCapture
* @package app\components\taskqueue\jobs
*/
class TaskCapture implements JobInterface
{
    protected array $taskIds;

    /**
     * TaskCapture constructor.
     * @param array $taskIds
```

```
     */
    public function __construct(array $taskIds)
    {
        $this->taskIds = $taskIds;
    }

    /**
     * @return bool
     */
    public function process(): bool
    {
        foreach ($this->taskIds as $taskId) {
            $this->runTask((int)$taskId);
        }
        return true;
    }

    /**
     * @param int $taskId
     */
    private function runTask(int $taskId)
    {
        $command = "php yii task-implementation/implementation $taskId >
/dev/null 2> /dev/null & echo $!";
        exec($command,$output, $returnCode);
    }

}


components/taskqueue/jobs/TaskImplementation.php
<?php
```

```
namespace app\components\taskqueue\jobs;

use app\models\tasks\Task;
use app\models\tasks\TaskImplementation as ImplementationLog;
use yii\httpclient\Client;
use Exception;

/**
* Class TaskImplementation
* @package app\components\taskqueue\jobs
*/
class TaskImplementation implements JobInterface
{
  private Task $task;
  private ImplementationLog $implementationLog;

  /**
   * TaskImplementation constructor.
   * @param Task $task
   */
  public function __construct(Task $task)
  {
     $this->task = $task;
     $this->implementationLog = new ImplementationLog();
  }

  /**
   * @return bool
   * @throws \Throwable
   * @throws \yii\db\StaleObjectException
   */
```

```
public function process(): bool
{
    $this->task->setScenario(Task::SCENARIO_IN_PROCESS);
    if (!$this->task->canBeProcessed()) {
        throw new Exception("Can not processing. [taskId={$this->task->id}]");
    }

    return $this->request();
}

/**
 * @return bool
 * @throws \Throwable
 * @throws \yii\base\InvalidConfigException
 * @throws \yii\db\StaleObjectException
 * @throws \yii\httpclient\Exception
 */
private function request()
{
    $request = (new Client())->createRequest()
        ->setMethod($this->task->type->method)
        ->setUrl($this->task->type->path)
        ->setData($this->task->getData());

    $this->saveRequest((string)$request);

    $response = $request->send ();
    if (!$response->isOk) {
        $this->saveResponse((string)$response);
        $this->processFailure();
        return false;
    }
```

```
        $this->saveResponse((string)$response);

        $this->processSuccess();
        return true;
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    private function processFailure()
    {
        $this->task->processFail();
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    private function processSuccess()
    {
        $this->task->processSuccess();
    }

    /**
     * @param string $data
     */
    private function saveRequest(string $data)
    {
        $this->implementationLog->saveRequest($this->task, $data);
    }
```

```
    private function saveResponse(?string $data)
    {
        $this->implementationLog->saveResponse($data);
    }

}
```

components/taskqueue/workers/WorkerInterface.php

```
<?php

namespace app\components\taskqueue\workers;

use app\components\taskqueue\jobs\JobInterface;

/**
* Interface WorkerInterface
* @package app\components\taskqueue\workers
*/
interface WorkerInterface
{
    /**
     * WorkerInterface constructor.
     * @param JobInterface $task
     */
    public function __construct(JobInterface $task);

    /**
     * @return bool
     */
    public function process(): bool;
}
```

components/taskqueue/workers/Worker.php

```
<?php

namespace app\components\taskqueue\workers;

use app\components\taskqueue\jobs\JobInterface;

/**
* Class Worker
* @package app\components\taskqueue\workers
*/
class Worker implements WorkerInterface
{
  private JobInterface $job;

  /**
   * Worker constructor.
   * @param JobInterface $job
   */
  public function __construct(JobInterface $job)
  {
     $this->job = $job;
  }

  /**
   * @return bool
   */
  public function process(): bool
  {
     return $this->job->process();
```

```
    }

}
```

components/taskqueue/workers/WorkerFactory.php

```
<?php

namespace app\components\taskqueue\workers;

use app\components\taskqueue\jobs\TaskArchiving;
use app\components\taskqueue\jobs\TaskCapture;
use app\components\taskqueue\jobs\TaskImplementation;
use app\models\tasks\Task;
use Exception;

/**
* Class WorkerFactory
* @package app\components\taskqueue\workers
*/
class WorkerFactory
{
    /**
     * @param int $workId
     * @param int $taskId
     * @return WorkerInterface|null
     * @throws \yii\db\Exception
     */
    public static function getTaskImplementation(int $workId, int $taskId):
?WorkerInterface
    {
        $task = Task::captureTask($taskId, $workId);
```

```
    if (!$task) {
        throw new Exception("Can not capture task. [taskId=$taskId]");
    }

    return new Worker(new TaskImplementation($task));
}

/**
 * @param int $workId
 * @return WorkerInterface|null
 * @throws \yii\db\Exception
 */
public static function getTaskCapture(int $workId): ?WorkerInterface
{
    $taskIds = Task::capture($workId);
    if (empty($taskIds)) {
        return null;
    }

    return new Worker(new TaskCapture($taskIds));
}

/**
 * @return WorkerInterface|null
 */
public static function getTaskArchiving(): ?WorkerInterface
{
    $tasks = Task::getArchivingTasks(TaskArchiving::ARCHIVING_TIME);
    if (empty($tasks)) {
        return null;
    }
```

```
            return new Worker(new TaskArchiving($tasks));
        }
    }
```

controllers/TaskController.php

```
<?php

namespace app\controllers;

use yii\filters\auth\HttpBasicAuth;
use yii\rest\ActiveController;
use yii\web\Response;

/**
* Class TaskController
*
* GET /tasks: получение постранично списка всех задач;
* Example: curl --user userName: -i "http://tq.local/tasks"
*
* POST /tasks: создание новой задачи;
* Example: curl --user userName: -i -H "Content-Type: application/json" -X POST "http://tq.local/tasks" -d '{"typeId":1,"statusId":1}'
*
* GET /tasks/123: получение информации по конкретной задаче с id равным 123;
* Example: curl --user userName: -i -X GET "http://tq.local/tasks/123"
*
* PUT /tasks/123: изменение информации по конкретной задаче с id равным 123;
* Example: curl --user userName: -i -H "Content-Type: application/json" -X PUT "http://tq.local/tasks/30" -d '{"typeId":100}'
```

```
*
* @package app\controllers
*/
class TaskController extends ActiveController
{
  public $modelClass = 'app\models\tasks\Task';

  /**
   * @return array|array[]
   */
  public function behaviors(): array
  {
     $behaviors = parent::behaviors();
     $behaviors['contentNegotiator']['formats'] = [
        'application/json' => Response::FORMAT_JSON]
     ;
     $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
     ];

     return $behaviors;
  }

  /**
   * @return array
   */
  public function actions(): array
  {
     $actions = parent::actions();

     unset($actions['delete']);
```

```
            return $actions;
        }


    }

    controllers/TaskTypeController.php
    <?php

    namespace app\controllers;

    use yii\filters\auth\HttpBasicAuth;
    use yii\rest\ActiveController;
    use yii\web\Response;

    /**
    * Class TaskTypeController
    *
    * GET /tasks: получение постранично списка всех типов задач;
    * Example: curl --user userName: -i "http://tq.local/task-types"
    *
    * POST /tasks: создание нового типа задачи;
    * Example: curl --user userName: -i -H "Content-Type: application/json" -X
POST "http://tq.local/task-types" -d
'{"path":"url","name":"name","priority":100,"cntAttempts":3}'
    *
    * GET /tasks/123: получение информации по конкретному типу задачи с id
равным 123;
    * Example: curl --user userName: -i -X GET "http://tq.local/task-types/123"
    *
    * PUT /tasks/123: изменение информации по конкретному типу задачи с id
равным 123;
```

```
    * Example: curl --user userName: -i -H "Content-Type: application/json" -X
PUT "http://tq.local/task-types/30" -d '{"priority":10}'
    * @package app\controllers
    */
    class TaskTypeController extends ActiveController
    {
      public $modelClass = 'app\models\tasks\TaskType';

      /**
       * @return array|array[]
       */
      public function behaviors(): array
      {
        $behaviors = parent::behaviors();
        $behaviors['contentNegotiator']['formats'] = [
          'application/json' => Response::FORMAT_JSON]
        ;
        $behaviors['authenticator'] = [
          'class' => HttpBasicAuth::class,
        ];

        return $behaviors;
      }

      /**
       * @return array
       */
      public function actions(): array
      {
        $actions = parent::actions();

        unset($actions['delete']);
```

```
        return $actions;
    }
}
```

models/tasks/Task.php

```
<?php

namespace app\models\tasks;

use Yii;
use yii\db\ActiveRecord;
use yii\db\Expression;
use Exception;

/**
* Class Task
* @property int $id
* @property int $typeId
* @property int $statusId
* @property int $workerId
* @property string $data
* @property string $createTime
* @property string $updateTime
* @property string $finishTime
* @property string $waitToTime
* @property int $cntUnsuccessfulAttempts
*
* @property TaskType $type
* @property TaskImplementation[] $implementations
*
```

```
     * @package app\models\tasks
     */
     class Task extends ActiveRecord
     {
        const TABLE_NAME = 'task';

        const FIELD_ID = 'id';
        const FIELD_TYPE_ID = 'typeId';
        const FIELD_STATUS_ID = 'statusId';
        const FIELD_WORKER_ID = 'workerId';
        const FIELD_DATA = 'data';
        const FIELD_CREATE_TIME = 'createTime';
        const FIELD_UPDATE_TIME = 'updateTime';
        const FIELD_FINISH_TIME = 'finishTime';
        const FIELD_WAIT_TO_TIME = 'waitToTime';
        const FIELD_CNT_UNSUCCESSFUL_ATTEMPTS =
'cntUnsuccessfulAttempts';

        const SCENARIO_IN_PROCESS = 'inProcess';

        const FAIL_WAITE_TIME = 60;

        /**
         * @param $taskId
         * @param int $workerId
         * @return static|null
         * @throws \yii\db\Exception
         */
        public static function captureTask($taskId, int $workerId): ?self
        {
           $sql = <<<SQL
     UPDATE task SET statusId=:inProgress, workerId=:workerId
```

```
WHERE statusId=:capture AND id=:taskId
SQL;

        $isCaptured = Yii::$app->db->createCommand(
            $sql,
            [
                ':inProgress' => TaskStatus::STATUS_IN_PROGRESS,
                ':capture' => TaskStatus::STATUS_CAPTURE,
                ':workerId' => $workerId,
                ':taskId' => $taskId,
            ]
        )->execute();

        if (!$isCaptured) {
            return null;
        }

        return static::findOne($taskId);
    }

    /**
     * @param int $workerId
     * @return array|\yii\db\DataReader
     * @throws \yii\db\Exception
     */
    public static function capture(int $workerId)
    {
        $sql = <<<SQL
SELECT min(T.id) as id
FROM task as T
JOIN taskType as TT ON T.typeId=TT.id
```

```
        WHERE T.workerId IS NULL AND (T.statusId in(:todo) OR (T.statusId IN (:fail)
AND T.cntUnsuccessfulAttempts<TT.cntAttempts AND T.waitToTime<=NOW()))
        AND NOT EXISTS (
            SELECT id FROM task as T2
            WHERE T.typeId=T2.typeId AND (T2.statusId NOT IN (:todo, :done, :fail,
:postponed) OR (T2.statusId=:fail AND
(T2.cntUnsuccessfulAttempts>=TT.cntAttempts  OR T2.waitToTime>NOW())))
         )
        GROUP BY T.typeId
        ORDER BY TT.priority DESC
        LIMIT 100
        SQL;

            $taskIds = \Yii::$app->db->createCommand(
               $sql,
               [
                  ':todo'=> TaskStatus::STATUS_TODO,
                  ':done'=> TaskStatus::STATUS_DONE,
                  ':fail'=> TaskStatus::STATUS_FAIL,
                  ':postponed'=> TaskStatus::STATUS_POSTPONED,
               ]
            )->queryColumn();

            if (!empty($taskIds)) {
               Yii::$app->db->createCommand()->update(
                  static::TABLE_NAME,
                  ['workerId' => $workerId, 'statusId' =>
TaskStatus::STATUS_CAPTURE],
                  ['in', 'id', $taskIds]
               )->execute();
            }
```

```
        return $taskIds;
    }

    /**
     * @param int $archivingTime
     * @return static[]
     */
    public static function getArchivingTasks(int $archivingTime)
    {
        return static::find()
            ->where([static::FIELD_STATUS_ID => TaskStatus::STATUS_DONE])
            ->andWhere(['<', static::FIELD_FINISH_TIME, new
Expression("NOW() - INTERVAL $archivingTime SECOND")])
            ->limit(100)
            ->all();
    }


    /**
     * @return string
     */
    public static function tableName(): string
    {
        return '{{' . static::TABLE_NAME . '}}';
    }

    /**
     * @return array
     */
    public function rules(): array
    {
```

```
            return [
                [[static::FIELD_TYPE_ID, static::FIELD_STATUS_ID], 'required',
'strict' => true],
                [[static::FIELD_TYPE_ID, static::FIELD_STATUS_ID], 'integer'],
                [static::FIELD_CNT_UNSUCCESSFUL_ATTEMPTS, 'integer', 'except'
=> static::SCENARIO_IN_PROCESS],
                [static::FIELD_TYPE_ID, 'in', 'range' =>
array_keys(TaskType::getList())],
                [static::FIELD_STATUS_ID, 'in', 'range' =>
array_keys(TaskStatus::getList())],
                [static::FIELD_DATA, 'trim'],
            ];
        }

        /**
         * @return array
         */
        public function fields(): array
        {
            return [
                static::FIELD_TYPE_ID,
                static::FIELD_STATUS_ID,
                static::FIELD_DATA,
                static::FIELD_CREATE_TIME,
                static::FIELD_UPDATE_TIME,
                static::FIELD_FINISH_TIME,
                static::FIELD_WAIT_TO_TIME,
                static::FIELD_CNT_UNSUCCESSFUL_ATTEMPTS,
            ];
        }
```

```
        /**
         * @return \yii\db\ActiveQuery
         */
        public function getType()
        {
            return $this->hasOne(TaskType::class, [TaskType::FIELD_ID =>
static::FIELD_TYPE_ID]);
        }

        /**
         * @return \yii\db\ActiveQuery
         */
        public function getImplementations()
        {
            return $this->hasMany(TaskImplementation::class,
[TaskImplementationArchive::FIELD_TASK_ID => static::FIELD_ID]);
        }

        /**
         * @return mixed
         */
        public function getData()
        {
            return (string)$this->data;
        }

        /**
         * @return bool
         */
        public function canBeProcessed(): bool
        {
            return $this->statusId == TaskStatus::STATUS_IN_PROGRESS;
```

```
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    public function processSuccess()
    {
        $this->finishTime = new Expression('NOW()');
        $this->setStatus(TaskStatus::STATUS_DONE);
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    public function processFail()
    {
        $this->workerId = null;
        $this->waitToTime = new Expression("NOW() + INTERVAL {$this->getFailWaitTime()} SECOND");
        $this->incrementUnsuccessfulAttempts();
        $this->setStatus(TaskStatus::STATUS_FAIL);
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\StaleObjectException
     */
    public function processError()
    {
        $this->incrementUnsuccessfulAttempts();
```

```
            $this->setStatus(TaskStatus::STATUS_ERROR);
        }

        /**
         * @param int $statusId
         * @throws \Throwable
         * @throws \yii\db\StaleObjectException
         */
        public function setStatus(int $statusId)
        {
            $this->statusId = $statusId;
            if (!$this->update()) {
                throw new Exception("Can not update status. [taskId={$this->id}]");
            }
        }

        /**
         * @throws \Throwable
         * @throws \yii\db\StaleObjectException
         */
        public function moveToArchive()
        {
            if ($this->statusId != TaskStatus::STATUS_DONE) {
                throw new Exception("Current task can not moved to archive. [id=$this->id]");
            }

            $transaction = Yii::$app->db->beginTransaction();
            try {
                $archive = new TaskArchive($this->getAttributes());
                $archive->id = null;
                $archive->taskId = $this->id;
```

```
            if (!$archive->save()) {
                throw new Exception("Can not created archive. [id=$this->id]");
            }
            foreach ($this->implementations as $implementation) {
                $implementation->moveToArchive();
            }

            $this->delete();

            $transaction->commit();
        } catch (Exception $e) {
            $transaction->rollBack();
            throw $e;
        }
    }

    /**
     * @return int
     */
    private function getFailWaitTime(): int
    {
        return static::FAIL_WAITE_TIME;
    }

    private function incrementUnsuccessfulAttempts()
    {
        $this->cntUnsuccessfulAttempts = new
Expression('cntUnsuccessfulAttempts + 1');
    }
}
```

models/tasks/TaskArchive.php

```
<?php

namespace app\models\tasks;

use yii\db\ActiveRecord;

/**
* Class Task
* @property int $id
* @property int $taskId
* @property int $typeId
* @property int $statusId
* @property int $workerId
* @property string $data
* @property string $createTime
* @property string $updateTime
* @property string $finishTime
* @property int $cntUnsuccessfulAttempts
*
* @package app\models\tasks
*/
class TaskArchive extends ActiveRecord
{
   const TABLE_NAME = 'taskArchive';

   const FIELD_ID = 'id';
   const FIELD_TASK_ID = 'taskId';
   const FIELD_TYPE_ID = 'typeId';
   const FIELD_STATUS_ID = 'statusId';
   const FIELD_WORKER_ID = 'workerId';
   const FIELD_DATA = 'data';
```

```
    const FIELD_CREATE_TIME = 'createTime';
    const FIELD_UPDATE_TIME = 'updateTime';
    const FIELD_FINISH_TIME = 'finishTime';
    const FIELD_CNT_UNSUCCESSFUL_ATTEMPTS =
'cntUnsuccessfulAttempts';


    /**
     * @return string
     */
    public static function tableName(): string
    {
        return '{{' . static::TABLE_NAME . '}}';
    }
}


models/tasks/TaskImplementation.php
<?php

namespace app\models\tasks;

use yii\db\ActiveRecord;
use Exception;
use Yii;

/**
* Class TaskImplementation
* @property int $id
* @property int $taskId
* @property string $rawRequest
* @property string $rawResponse
* @property string $createTime
```

```
* @property int $workTime
*
* @package app\models\tasks
*/
class TaskImplementation extends ActiveRecord
{
   const TABLE_NAME = 'taskImplementation';

   const FIELD_ID = 'id';
   const FIELD_TASK_ID = 'taskId';
   const FIELD_RAW_REQUEST = 'rawRequest';
   const FIELD_RAW_RESPONSE = 'rawResponse';
   const FIELD_CREATE_TIME = 'createTime';
   const FIELD_WORK_TIME = 'workTime';

   private float $requestTime;


   /**
    * @return string
    */
   public static function tableName(): string
   {
      return '{{' . static::TABLE_NAME . '}}';
   }

   /**
    * @param Task $task
    * @param $data
    */
   public function saveRequest(Task $task, string $data)
   {
```

```
        $this->requestTime = microtime(true);

        $this->taskId = $task->id;
        $this->rawRequest = $data;

        $this->save();
    }

    /**
     * @param $data
     */
    public function saveResponse(?string $data)
    {
        if ($this->requestTime) {
            $this->workTime = microtime(true) - $this->requestTime;
        }
        $this->rawResponse = $data;

        $this->save();
    }

    /**
     * @throws \Throwable
     * @throws \yii\db\Exception
     * @throws \yii\db\StaleObjectException
     */
    public function moveToArchive()
    {
        $transaction = Yii::$app->db->beginTransaction();
        try {
            $archive = new TaskImplementationArchive($this->getAttributes());
            $archive->id = null;
```

```
            $archive->taskImplementationId = $this->id;

            if (!$archive->save()) {
                throw new Exception("Can not created archive. [id=$this->id]");
            }
            $this->delete();

            $transaction->commit();
        } catch (Exception $e) {
            $transaction->rollBack();
            throw $e;
        }
    }
}
```

models/tasks/TaskImplementationArchive.php

```
<?php

namespace app\models\tasks;

use yii\db\ActiveRecord;

/**
* Class TaskImplementationArchive
* @property int $id
* @property int $taskImplementationId
* @property int $taskId
* @property string $rawRequest
* @property string $rawResponse
* @property string $createTime
* @property int $workTime
```

```
*
* @package app\models\tasks
*/
class TaskImplementationArchive extends ActiveRecord
{
  const TABLE_NAME = 'taskImplementationArchive';

  const FIELD_ID = 'id';
  const FIELD_TASK_IMPLEMENTATION_ID = 'taskImplementationId';
  const FIELD_TASK_ID = 'taskId';
  const FIELD_RAW_REQUEST = 'rawRequest';
  const FIELD_RAW_RESPONSE = 'rawResponse';
  const FIELD_CREATE_TIME = 'createTime';
  const FIELD_WORK_TIME = 'workTime';

  /**
   * @return string
   */
  public static function tableName(): string
  {
     return '{{' . static::TABLE_NAME . '}}';
  }
}
```

models/tasks/TaskStatus.php

```
<?php

namespace app\models\tasks;

use yii\db\ActiveRecord;
use yii\helpers\ArrayHelper;
```

```
/**
* Class TaskStatus
* @property int $id
* @property string $name
*
* @package app\models\tasks
*/
class TaskStatus extends ActiveRecord
{
   const TABLE_NAME = 'taskStatus';

   const FIELD_ID = 'id';
   const FIELD_NAME = 'name';

   const STATUS_TODO = 1;
   const STATUS_CAPTURE = 2;
   const STATUS_IN_PROGRESS = 3;
   const STATUS_DONE = 4;
   const STATUS_FAIL = 5;
   const STATUS_ERROR = 6;
   const STATUS_POSTPONED = 6;


   /**
    * @return string
    */
   public static function tableName(): string
   {
      return '{{' . static::TABLE_NAME . '}}';
   }
```

```
    /**
     * @return array
     */
    public static function getList(): array
    {
        return ArrayHelper::map(static: :find()->asArray()->all(),
static::FIELD_ID, static::FIELD_NAME);
    }
}
```

models/tasks/TaskType.php

```
<?php

namespace app\models\tasks;

use yii\db\ActiveRecord;
use yii\helpers\ArrayHelper;

/**
* Class TaskType
* @property int $id
* @property string $path
* @property string $method
* @property string $name
* @property int $priority
* @property int $cntAttempts
*
* @package app\models\tasks
*/
class TaskType extends ActiveRecord
{
```

```
    const TABLE_NAME = 'taskType';

    const FIELD_ID = 'id';
    const FIELD_PATH = 'path';
    const FIELD_METHOD = 'method';
    const FIELD_NAME = 'name';
    const FIELD_PRIORITY = 'priority';
    const FIELD_CNT_ATTEMPTS = 'cntAttempts';


    /**
     * @return string
     */
    public static function tableName(): string
    {
        return '{{' . static::TABLE_NAME . '}}';
    }

    /**
     * @return array
     */
    public static function getList(): array
    {
        return ArrayHelper::map(static::find()->asArray()->all(),
static::FIELD_ID, static::FIELD_NAME);
    }

    /**
     * @return array
     */
    public function rules(): array
    {
```

```
            return [
                [[static::FIELD_PATH, static::FIELD_NAME,
static::FIELD_PRIORITY, static::FIELD_CNT_ATTEMPTS], 'required', 'strict'],
                [[static::FIELD_CNT_ATTEMPTS, static::FIELD_PRIORITY],
'integer'],
                [[static::FIELD_PATH, static::FIELD_NAME], 'string', 'length' => [3,
255]],
                [static::FIELD_METHOD, 'in', 'range' => ['GET', 'POST']],
                [static::FIELD_METHOD, 'default', 'value' => 'POST'],
                [[static::FIELD_PATH, static::FIELD_NAME], 'trim'],
            ];
        }
    }
```

models/worklog/Log.php

```
<?php

namespace app\models\worklog;

use yii\db\ActiveRecord;

/**
* Class Log
*
* @property int $id
* @property int $typeId
* @property int $statusId
* @property string $startTime
* @property string $finishTime
* @property int $cnt
* @property WorkType|null $workType
```

```
*
* @package app\models\worklog
*/
class Log extends ActiveRecord
{
  const TABLE_NAME = 'workLog';

  const FIELD_ID = 'id';
  const FIELD_TYPE_ID = 'typeId';
  const FIELD_STATUS_ID = 'statusId';
  const FIELD_START_TIME = 'startTime';
  const FIELD_FINISH_TIME = 'finishTime';
  const FIELD_CNT = 'cnt';


  /**
   * @return string
   */
  public static function tableName(): string
  {
     return '{{' . static::TABLE_NAME . '}}';
  }

  /**
   * @param WorkType $workType
   * @return static|null
   * @throws \Exception
   */
  public static function createInstance(WorkType $workType): ?self
  {
     $typeId = $workType->id;
     $cntStream = $workType->maxStream;
```

```
            $statusId = LogStatus::IN_PROGRESS;

            $query = <<<SQL
        INSERT INTO workLog (typeId, statusId)
        SELECT *
        FROM (SELECT {$typeId} as typeId, {$statusId} as statusId)  as temp
        WHERE NOT EXISTS (SELECT COUNT(id) AS cntStream FROM workLog
WHERE typeId = {$typeId} AND statusId = {$statusId} HAVING cntStream >=
{$cntStream})
        SQL;

            $isAdded = \Yii::$app->db->createCommand($query, [])->execute();
            if (!$isAdded) {
               return null;
            }

            $logId = \Yii::$app->db->getLastInsertID();

            if (!($instance = static::findOne($logId))) {
               throw new \Exception("Cannot create log. [workTypeId=$workType-
>id]");
            }

            return $instance;
         }

         /**
          * @return \yii\db\ActiveQuery
          */
         public function getWorkType()
         {
```

```
        return $this->hasOne(WorkType::class, [WorkType::FIELD_ID =>
static::FIELD_TYPE_ID]);
      }
    }
```

models/worklog/LogStatus.php

```
<?php

namespace app\models\worklog;

/**
* Class LogStatus
* @package app\models\worklog
*/
class LogStatus
{
  const IN_PROGRESS = 1;
  const DONE = 2;
  const FAILED = 3;
  const KILLED = 4;
}
```

models/worklog/WorkType.php

```
<?php

namespace app\models\worklog;

use yii\db\ActiveRecord;

/**
```

```
* Class WorkType
* @property int $id
* @property string $name
* @property int $maxTime
* @property int $maxStream
* @property int $waitTime
*
* @package app\models\worklog
*/
class WorkType extends ActiveRecord
{
    const TABLE_NAME = 'workType';

    const FIELD_ID = 'id';
    const FIELD_NAME = 'name';
    const FIELD_MAX_TIME = 'maxTime';
    const FIELD_MAX_STREAM = 'maxStream';
    const FIELD_WAIT_TIME = 'waitTime';

    const TYPE_TASK_CAPTURE = 1;
    const TYPE_TASK_IMPLEMENTATION = 2;
    const TYPE_TASK_ARCHIVING = 3;

    /**
     * @return string
     */
    public static function tableName(): string
    {
        return '{{' . static::TABLE_NAME . '}}';
    }

    /**
```

```
     * @param string $name
     * @return static
     */
    public static function findByName(string $name): self
    {
        return static::find()
            ->where([static::FIELD_NAME => $name])
            ->one();
    }
}
```

models/ApiLog.php

```
<?php

namespace app\models;

use Yii;
use yii\db\ActiveRecord;
use yii\web\Response;

/**
* Class ApiLog
*
* @property int $id
* @property string $url
* @property string $method
* @property string $authCredentials
* @property string $userIp
* @property string $headers
* @property string $rawBody
* @property int $responseStatus
```

```
* @property string $createTime
*
* @package app\models
*/
class ApiLog extends ActiveRecord
{
    const TABLE_NAME = 'apiLog';

    const FIELD_ID = 'id';
    const FIELD_URL = 'url';
    const FIELD_METHOD = 'method';
    const FIELD_AUTH_CREDENTIALS = 'authCredentials';
    const FIELD_USER_IP = 'userIp';
    const FIELD_HEADERS = 'headers';
    const FIELD_RAW_BODY = 'rawBody';
    const FIELD_RESPONSE_STATUS = 'responseStatus';
    const FIELD_CREATE_TIME = 'createTime';

    /**
     * @return string
     */
    public static function tableName(): string
    {
        return '{{' . static::TABLE_NAME . '}}';
    }

    /**
     * @param Response $response
     */
    public static function log(Response $response)
    {
        $request = Yii::$app->request;
```

```
        try {
            $log = new static();
            $log->url = $request->getAbsoluteUrl();
            $log->method = $request->getMethod();
            $log->authCredentials = json_encode($request->getAuthCredentials());
            $log->userIp = $request->getUserIP();
            $log->headers = json_encode($request->getHeaders()->toArray());
            $log->rawBody = $request->getRawBody();
            $log->responseStatus = (int)$response->getStatusCode();
            $log->save();

        } catch (\Throwable $e) {
            Yii::error($e->getMessage());
        }
    }
}
```

models/User.php

```
<?php

namespace app\models;

/**
* Class User
* @package app\models
*/
class User extends \yii\base\BaseObject implements \yii\web\IdentityInterface
{
    public $id;
    public $username;
    public $password;
    public $authKey;
```

```
    public $accessToken;

    private static $users = [
        '100' => [
            'id' => '100',
            'username' => 'admin',
            'password' => 'admin',
            'authKey' => 'test100key',
            'accessToken' => '100-token',
        ],
        '101' => [
            'id' => '101',
            'username' => 'demo',
            'password' => 'demo',
            'authKey' => 'test101key',
            'accessToken' => '101-token',
        ],
    ];


    /**
     * {@inheritdoc}
     */
    public static function findIdentity($id)
    {
        return isset(self::$users[$id]) ? new static(self::$users[$id]) : null;
    }

    /**
     * {@inheritdoc}
     */
    public static function findIdentityByAccessToken($token, $type = null)
```

```
{
    foreach (self::$users as $user) {
        if ($user['accessToken'] === $token) {
            return new static($user);
        }
    }

    return null;
}

/**
 * Finds user by username
 *
 * @param string $username
 * @return static|null
 */
public static function findByUsername($username)
{
    foreach (self::$users as $user) {
        if (strcasecmp($user['username'], $username) === 0) {
            return new static($user);
        }
    }

    return null;
}

/**
 * {@inheritdoc}
 */
public function getId()
{
```

```
    return $this->id;
}

/**
 * {@inheritdoc}
 */
public function getAuthKey()
{
    return $this->authKey;
}

/**
 * {@inheritdoc}
 */
public function validateAuthKey($authKey)
{
    return $this->authKey === $authKey;
}

/**
 * Validates password
 *
 * @param string $password password to validate
 * @return bool if password provided is valid for current user
 */
public function validatePassword($password)
{
    return $this->password === $password;
}
```