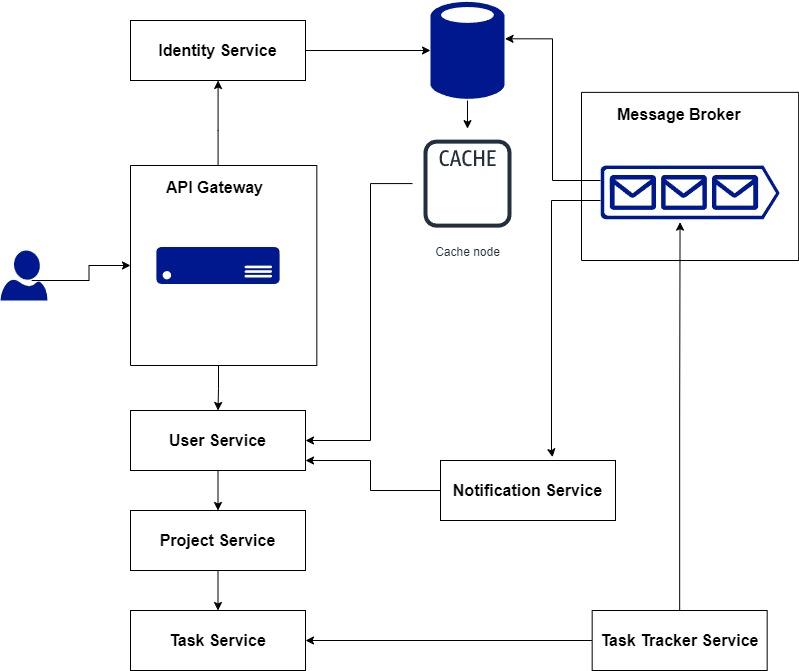
## Дизайн веб-приложения для управления задачами (Task Manager)

****

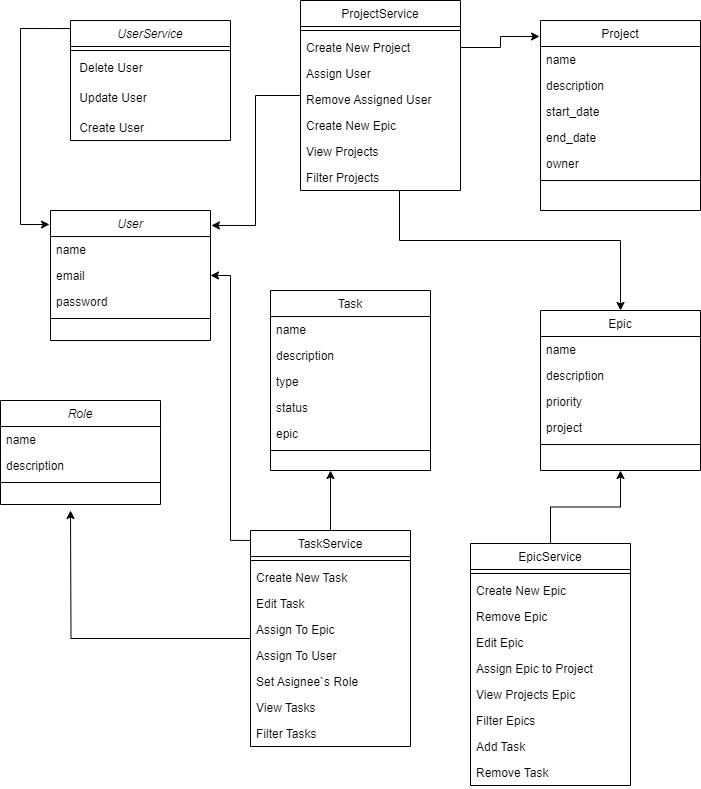
**SPoF:**

1. Api Gateway => добавить дополнительные сервера + балансировщики нагрузки
2. База данных => репликация
3. Message Broker => добавить инстансы

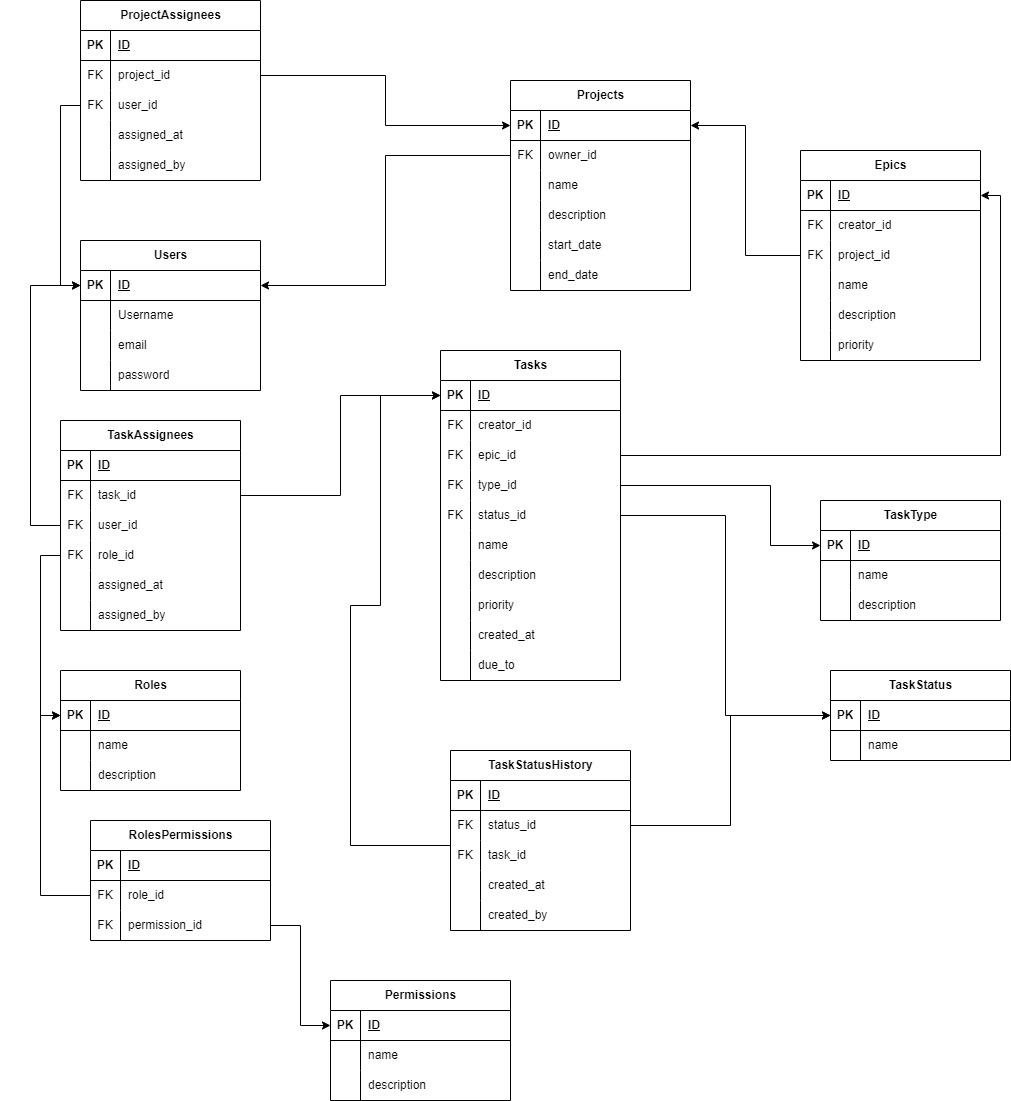
**Список сущностей, классов, моделей и связи:**

1. **Пользователи:**
   * Класс: User
   * Модель: user\_id, username, email, password, role\_id
2. **Роли пользователей:**
   * Класс: Role
   * Модель: role\_id, name
3. **Разрешения**:
   * Класс Permissions
   * Модель: permission\_id, name, description
4. **Проекты:**
   * Класс: Project
   * Модель: project\_id, name, description, owner\_id, start\_date, end\_date
5. **Эпики:**
   * Класс: Epic
   * Модель: epic\_id, creator\_id, name, description, project\_id, priority
6. **Задачи:**
   * Класс: Task
   * Модель: task\_id, task\_type\_id, task\_status\_id, epic\_id, creator\_id, name, description, priority, due\_date
7. **Статус задачи:**
   * Класс: TaskStatus
   * Модель: task\_status\_id, name
8. **Тип задачи:**
   * Класс: TaskType
   * Модель: task\_type\_id, name, description

### Схема взаимозависимостей классов



**Схема отношений в бд**



**Пользовательские сценарии:**

* **Создание проекта**
  1. Пользователь создает новый проект
  2. Вводит название, описание, задает сроки выполнения
  3. Выбирает участников
* **Создание эпика**
  1. Пользователь выбирает существующий проект
  2. Создает новый эпик в рамках проекта
  3. Вводит название, описание
* **Создание задачи:**
  1. Пользователь выбирает проект и эпик.
  2. Вводит название, описание, приоритет и срок выполнения задачи.
  3. Выбирает исполнителя
  4. Назначает других участников задачи и задает им роли
* **Просмотр списка задач:**
  1. Пользователь выбирает проект.
  2. Просматривает список задач в выбранном проекте.
  3. Может фильтровать задачи по исполнителю, статусу, приоритету и сроку выполнения.
* **Редактирование задачи:**
  1. Пользователь открывает задачу.
  2. Меняет название, описание, приоритет
  3. Меняет список исполнителей и участников и их роли

**Решение проблемных кейсов в системе управления версиями**

Дано: высоконагруженный проект, в котором заняты ~15 разработчиков, ревьюеры и техлид. Разработка ведется в трех средах в системе управления версиями Git:

1. Features

1. Development

2. Pre-production

3. Production

Новый функционал добавляется в каждую среду последовательно, проходит тестирование и только после этого заливается в следующую среду. Среды отличаются друг от друга минорно, например, из-за хотфиксов, отправившихся сразу на этап Pre-production или Production, минуя этап Development.

**Проблема №1**: возникла критическая уязвимость в текущей системе авторизации, решить нужно очень оперативно, заменив полностью используемый вариант. Сложность в том, что текущая система авторизации применяется повсеместно и взаимодействует с другими рабочими системами.

**Задача**: определить, как максимально безопасно и в то же время эффективно использовать инструменты контроля версий (и возможно, автотестирования), чтобы эту уязвимость закрыть. Необходимо учесть, что в это же время на проекте разрабатываются и изменяются другие подсистемы, и готовые изменения нужно выливать сразу по мере готовности после прохождения ручного тестирования. Опиши, какие ветки нам понадобятся для решения задачи, откуда они будут брать начало и куда будут вливаться, а также как и кем будут решаться конфликты.

**Решение**:

1. **Создаем ветки:**
2. fix-auth**:** Создается из ветки features. В ней будет происходить разработка и тестирование новой системы авторизации.
3. hotfix-auth **(опционально):** Если необходимо срочно применить временное решение, создается ветка из production. В ней реализуется временный "хотфикс" для критической уязвимости.

**2. Разработка:**

1. fix-auth**:**
   * Разработчик создает новую ветку fix-auth из features.
   * Разработчик реализует новую систему авторизации в ветке fix-auth.
   * Тестировщики проводят ручное и автоматическое тестирование новой системы авторизации.
   * **Важно:** все изменения в других подсистемах должны происходить в ветке features параллельно с разработкой fix-auth.
2. hotfix-auth **(опционально):**
   * Если требуется срочный "хотфикс", разработчик создает ветку hotfix-auth из production.
   * В ветке hotfix-auth реализуется временное решение для критической уязвимости.
   * "Хотфикс" проходит ручное тестирование.
   * **Важно:** "хотфикс" не должен содержать функциональных изменений, только исправление уязвимости.

**3. Слияние:**

1. fix-auth**:**
   * После успешного тестирования ветка fix-auth сливается обратно в features.
   * **Важно:** перед слиянием необходимо тщательно проверить, чтобы изменения в fix-auth не конфликтовали с другими изменениями в features.
   * **Конфликты:**
     + Если конфликты возникают, их должен решать разработчик, который создал ветку fix-auth.
     + При необходимости будут привлекаться ревьюеры.
     + Для решения конфликтов могут использоваться инструменты Git, такие как git mergetool.
2. hotfix-auth **(опционально):**
   * После успешного тестирования "хотфикса" ветка hotfix-auth сливается обратно в production.
   * **Важно:** после слияния "хотфикса" необходимо как можно скорее начать разработку и тестирование новой системы авторизации в ветке fix-auth.
3. Техлид**:**
   * Контролирует процесс разработки и тестирования.
   * После успешного тестирования объединяет ветку features с веткой Pre-production.
   * Контролирует процесс развертывания на Pre-production.

**4. Развертывание:**

* **Новая система авторизации:**
  + Развертывается на Pre-production.
  + Проходит финальное тестирование.
  + После успешного тестирования разворачивается на Production.

**5. Автотестирование:**

* **Unit-тесты:**
  + Добавить unit-тесты для новой системы авторизации.
  + Запустить unit-тесты перед заливкой изменений в **development.**
* **Интеграционные тесты:**
  + Добавить интеграционные тесты для новой системы авторизации.
  + Запустить интеграционные тесты перед заливкой изменений в **pre-production.**
* **Нагрузочные тесты:**
  + Провести нагрузочное тестирование новой системы авторизации перед заливкой в production.
* Тесты должны запускаться автоматически при push в ветки fix-auth и hotfix-auth.
* **Feature flags:**
  + Если возможно, рассмотреть использование feature flags для поэтапного включения новой системы авторизации.

**6. Инструменты:**

* **GitLab CI/CD:**
  + Настройка автоматического запуска тестов при push в ветки fix-auth и hotfix-auth.
  + Оповещение команды о результатах тестов.
* **Git mergetool:**
  + Инструмент для визуального разрешения конфликтов при слиянии веток.

**7. Преимущества:**

* **Быстрое решение проблемы:**
  + Новая система авторизации будет разработана и развернута в кратчайшие сроки.
* **Минимальное влияние на другие подсистемы:**
  + Разработка новой системы авторизации будет вестись в изолированной ветке.
* **Контроль качества:**
  + Новая система авторизации будет проходить ручное и автоматическое тестирование.
* **Минимизация рисков:**
  + Конфликты будут решаться вручную опытными разработчиками.

**Проблема №2**: В процессе разработки возник конфликт между веткой Features, ответвленной от Production, и веткой Pre-production.

**Задача**: предложить способы решения таких конфликтов с минимальными затратами от ревьюера и разработчика.

**Решение**:

**1. Анализ конфликта:**

* **Определить причину:**
  + Выяснить, почему ветка Features была ответвлена от Production вместо Development.
  + Проанализировать изменения, внесенные в обе ветки.
* **Оценить масштаб:**
  + Определить количество и сложность конфликтов.
  + Оценить влияние конфликтов на функциональность системы.

**2. Стратегии решения:**

**2.1. Слияние:**

* **Использовать инструмент Git merge:**
  + Git merge может автоматически объединить изменения из обеих веток.
  + В случае конфликтов вручную решить их.
* **Слияние с помощью ребаза:**
  + Перебазировать изменения из ветки Features на Pre-production.
  + Решить конфликты, возникшие при ребазировании.

**2.2. Отмена изменений:**

* **Отменить изменения в ветке** Features**:**
  + Сбросить изменения в ветке Features к началу ветки Development.
  + Заново внести изменения, учитывая изменения в ветке Pre-production.

**3. Выбор стратегии:**

* **При небольших конфликтах:**
  + Использовать инструмент Git merge.
* **При сложных конфликтах:**
  + Использовать ребазинг.
* **При наличии критических ошибок:**
  + Отменить изменения в ветке Features.

**4. Предотвращение подобных конфликтов:**

* **Использовать стандартные ветки:**
  + Всегда ответвлять ветки Features от Development.
* **Проводить ревью кода:**
  + Ревьюер должен проверять правильность ветвления и потенциальные конфликты.
  + Ввести обязательное ревью кода перед созданием pull-request
* **Использовать инструменты CI/CD:**
  + CI/CD может автоматически проверять наличие конфликтов перед слиянием веток.