Часть 1. Проектирование архитектуры

1. Определение типа приложения

Разрабатываемая система представляет собой веб-игру-кликер, состоящую из двух основных частей: веб-приложения (backend) и клиентской части, реализованной на Unity. Выбор такого типа приложения обусловлен следующими факторами:

- **Разделение обязанностей**: Backend отвечает за обработку данных, хранение информации о пользователях и их прогрессе, а также предоставление API для клиента. Клиентская часть (Unity) обеспечивает игровой интерфейс и взаимодействие с backend для синхронизации данных.
- **Производительность и надежность**: Обработка игровых данных на серверной стороне повышает безопасность и предотвращает читерство. Unity обеспечивает высокую производительность и отзывчивость интерфейса.
- **Гибкость развертывания**: Backend разворачивается в Kubernetes, обеспечивая гибкость и масштабируемость. Клиентская часть распространяется как веб-приложение, доступное через браузер.
- **Масштабируемость**: Веб-архитектура позволяет легко добавлять новые функции и расширять игровую логику без изменения клиентской части. В дальнейшем возможно использование облачных технологий для балансировки нагрузки.

2. Выбор стратегии развёртывания

Для развертывания системы выбрана клиент-серверная архитектура с разделением на два уровня (2-tier architecture), что обеспечивает баланс между производительностью, масштабируемостью и простотой взаимодействия между компонентами:

- **Клиентская часть (Unity)** взаимодействует с сервером для получения данных о прогрессе пользователя и синхронизации игрового состояния.
- **Веб-приложение (backend)** обрабатывает запросы клиента, предоставляет API и хранит информацию о пользователях и их прогрессе в базе данных.

Преимущества выбранного развертывания:

- **Простота реализации и поддержки** – минимальное количество взаимодействующих компонентов снижает сложность разработки и тестирования.

- **Гибкость и возможность масштабирования** – в дальнейшем можно интегрировать балансировщики нагрузки или расширить систему до 3-уровневой архитектуры.
- **Высокая производительность** – клиентская часть (Unity) обеспечивает отзывчивый интерфейс, а серверная часть (backend) обрабатывает данные с минимальными задержками.
- **Централизованное хранение данных** – все данные пользователей хранятся на сервере, что обеспечивает безопасность и предотвращает потерю данных.

Возможности дальнейшего расширения:

При увеличении нагрузки и числа пользователей возможно реализовать репликацию и шардирование на стороне базы данных. Также можно использовать облачные технологии для балансировки нагрузки и кэширования запросов. Также есть возможность как горизонтального, так и вертикального масшабирования.

3. Обоснование выбора технологии

Для реализации системы выбраны следующие технологии:

- **Java (Spring Boot)** – для серверной части. Spring Boot обеспечивает высокую производительность, удобство разработки и поддержку асинхронных операций, что критично для работы с сетевыми запросами. Фреймворк поддерживает кроссплатформенность, что позволяет запускать серверную часть как в Windows-среде, так и на Linux-серверах.
- **PostgreSQL** – в качестве реляционной базы данных. PostgreSQL обеспечивает надежность, масштабируемость и поддержку сложных запросов, что важно для игровых систем.
- **Unity** – для клиентской части. Unity позволяет создавать интерактивные и визуально привлекательные интерфейсы для веб-игр, поддерживает интеграцию с веб-технологиями, что упрощает развертывание игры в браузере.
- **Kubernetes** – система оркестрации контейнеров, позволяющая гибко и автоматически разворачивать приложение.

Выбранные технологии обеспечивают баланс между производительностью и гибкостью, что особенно важно для проекта веб-игры-кликера. Spring Boot позволяет эффективно обрабатывать запросы, PostgreSQL – хранить и

управлять данными в надежном формате, а Unity – создавать современный и удобный игровой интерфейс.

4. Указание показателей качества (методология FURPS)

- **Functionality (Функциональные требования):**
  - Регистрация и аутентификация пользователей.
  - Хранение и синхронизация игрового прогресса.
  - Взаимодействие клиентской части с backend через API.
  - Поддержка мультимедийных элементов (графика, звук).
- **Usability (Удобство использования):**
  - Интуитивно понятный интерфейс.
  - Быстрая и удобная навигация.
  - Минимальное количество кликов для доступа к ключевому функционалу.
- **Reliability (Надёжность):**
  - Высокая готовность системы (90–99% времени безотказной работы).
  - Регулярное резервное копирование данных.
  - Обеспечение отказоустойчивости: при отказе отдельных компонентов система продолжает работать с ограниченным функционалом.
- **Performance (Производительность):**
  - Средняя задержка отклика не превышает 2 секунд.
  - Эффективное использование системных ресурсов.
  - Поддержка одновременного доступа большого количества пользователей.
  - Масштабируемость системы для роста нагрузки.
- **Supportability (Сопровождаемость):**
  - Возможность добавления нового функционала без значительных изменений существующего кода.
  - Централизованное логирование и обработка ошибок.
  - Автоматизированное резервное копирование.
  - Адаптивный дизайн для корректного отображения на различных устройствах.

5. Обозначение путей реализации сквозной функциональности

Для обеспечения надежности, безопасности и производительности системы реализованы следующие механизмы:

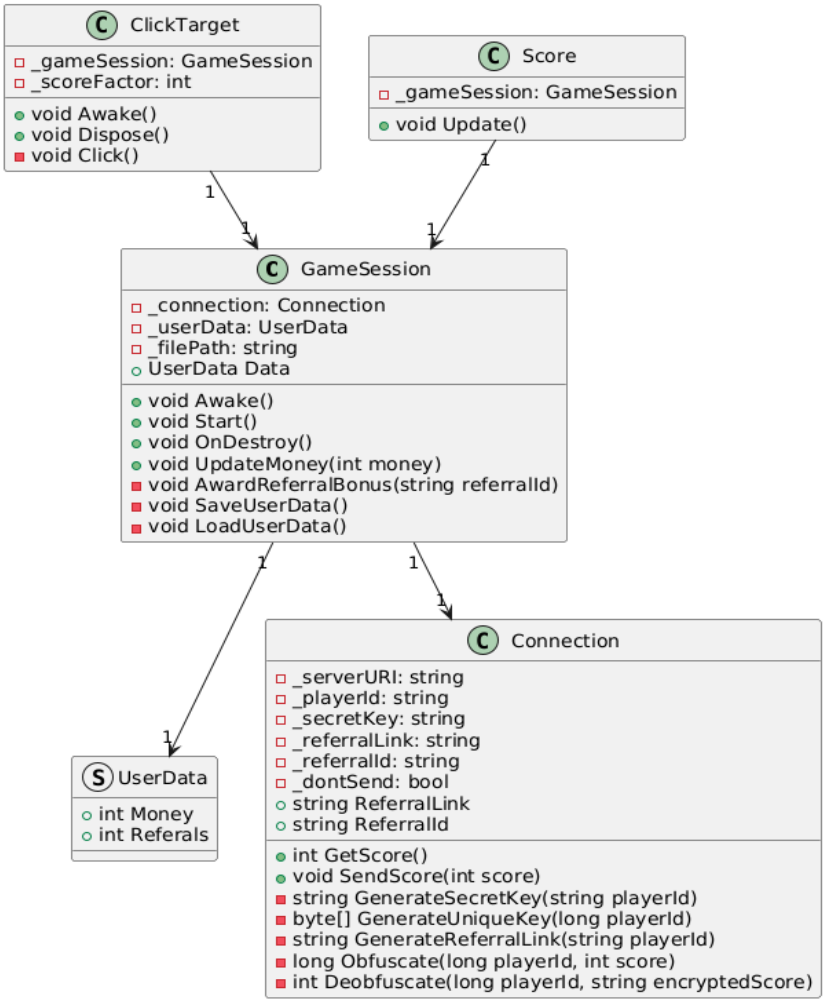
- **Аутентификация и авторизация**: Использование JWT-токенов для REST API, что обеспечивает централизованное управление доступом и безопасность.

- **Обработка исключений:** Единая система обработки ошибок, позволяющая централизованно регистрировать и обрабатывать исключительные ситуации, предотвращая утечки информации.
- **Кэширование:** Применение кэширования (с использованием Redis или Memcached) для снижения времени отклика и повышения производительности при работе с часто запрашиваемыми данными.
- **Безопасность:** Реализация мер защиты от CSRF, XSS, SQL-инъекций посредством встроенных механизмов Spring Boot и дополнительных модулей.
- **API GATEWAY:** Валидация доступа к микросервисам, также гибкое управление маршрутизацией к ним.

6. Изображение структурной схемы приложения

Ниже представлена структурная схема «Архитектура Unity части», отображающая основные взаимодействие классов в frontend части приложения:

Рис. 1. Структурная схема приложения (To Be):



Часть 2. Анализ архитектуры («As is»)

На данном этапе проведён анализ существующей архитектуры системы, сформированной в ходе первого Sprint Review. С использованием инструмента обратной инженерии (например, IBM Rational Rose) была сгенерирована диаграмма классов, отображающая структуру ключевых компонентов.

Рис. 2. Диаграмма классов (As is):

Примечание: Диаграмма демонстрирует основные модели системы:

Часть 3. Сравнение и рефакторинг

1. Сравнение архитектур «As is» и «To be»

- **Модульность и разделение ответственности:**
  В архитектуре «To be» наблюдается чёткое разделение между слоями (представление, бизнес-логика, доступ к данным), что облегчает масштабирование и сопровождение. В существующей архитектуре («As is») обнаружены случаи чрезмерной связности между модулями, что затрудняет внесение изменений.
- **Сквозная функциональность:**
  Проектируемая архитектура предусматривает централизованное управление аутентификацией, логированием, обработкой ошибок и кэшированием через middleware и специальные сервисы. В «As is» данные аспекты реализованы фрагментарно, что приводит к дублированию кода и усложнению поддержки.
- **Применение шаблонов проектирования:**
  Архитектура «To be» опирается на проверенные шаблоны (например, Model-View-Template, Repository, Service), что способствует улучшенной тестируемости и расширяемости. В существующей реализации подобные подходы используются не везде, что затрудняет масштабирование системы.

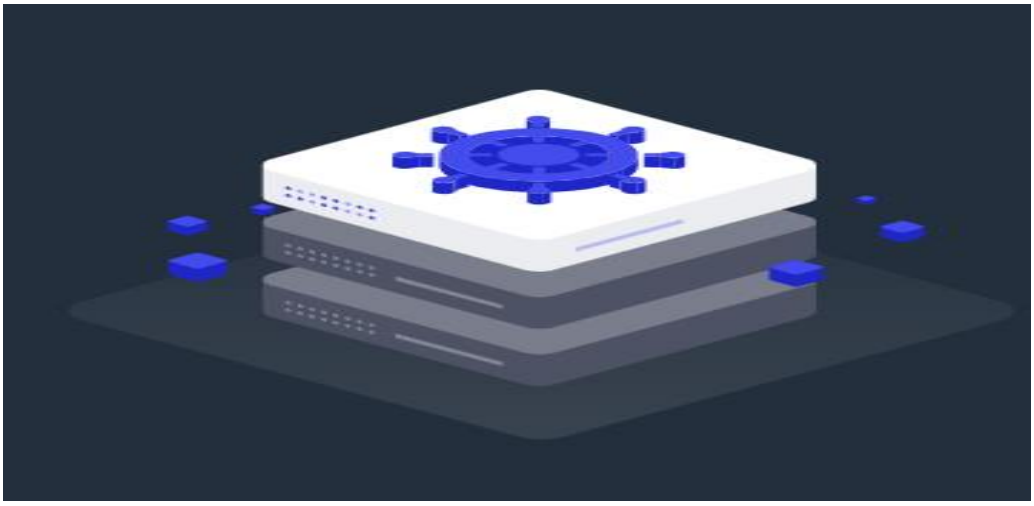
2. Анализ причин отличий

- **Эволюция системы:**
  Архитектура «As is» формировалась в ходе быстрого прототипирования, где приоритетом была скорость разработки, а не соблюдение принципов модульности и сквозной функциональности.
- **Ограниченные сроки:**
  На ранних этапах разработки не уделялось достаточное внимание внедрению единых решений для сквозных аспектов, что привело к фрагментарной реализации таких функций, как аутентификация и логирование.
- **Отсутствие чётких стандартов:**
  Недостаток использования единых архитектурных стандартов и шаблонов проектирования способствовал появлению рассогласованностей в структуре кода.

3. Пути улучшения архитектуры (Рефакторинг)

Для повышения качества архитектурного решения рекомендуется:

- **Внедрение CI/CD пайплайнов:** это позволит автоматически запускать тесты и разворачивать приложение.
- **Автомасштабирование K8s кластера в случае повышения нагрузки**
- **Внедрение Heathcheck на стороне K8s:** это позволит отслеживать состояние всей системы.
- **Централизированный сбор логов, а также дашборды:** позволит отслеживать состояние системы, а также находить узкие места.
- **Использование IaC инструментов, таких как Terraform и Ansible:** использование этих инструментов позволит описать всю инфраструктуру в нескольких файлах, что позволит грамотно распределить ресурсы, а также хранить конфигурацию продакшена на удаленном репозитории.
- **Load Balancing на стороне Cloud провайдера**
- **Использовать шаблоны проектирования:**
  Применять такие шаблоны, как Repository, Service и Dependency Injection для повышения тестируемости, снижения связности между компонентами и облегчения масштабирования системы.
- **Оптимизировать производительность:**
  Реализовать кэширование на уровне запросов к базе данных, оптимизировать запросы ORM и проводить регулярный аудит производительности для своевременного выявления и устранения узких мест.
- **Стандартизировать код и документировать архитектуру:**
  Разработать и внедрить стандарты кодирования, а также вести подробную документацию по архитектурным решениям, что позволит новым участникам проекта быстрее включаться в работу и обеспечит единообразие разработки.



Вывод

В результате проведённого исследования были выявлены основные различия между существующей («As is») и проектируемой («To be») архитектурами. В проектируемой архитектуре достигнуто лучшее разделение ответственности, централизовано реализована сквозная функциональность и заложены основы для масштабируемости и поддержки системы. Анализ существующего решения выявил слабые места, связанные с высокой связностью компонентов и фрагментарной реализацией критически важных функций. Предложенные пути рефакторинга, включая применение шаблонов проектирования и централизованное управление сквозными аспектами, позволят повысить надёжность, производительность и сопровождаемость системы.