**Контрольные вопросы**

1. **Что такое системы контроля версий (СКВ) и для решения каких задач они предназначаются?**

**Системы контроля версий (СКВ)** — это инструменты, которые помогают разработчикам управлять изменениями в файлах проекта. Они позволяют отслеживать изменения, сделанные в коде, сохранять историю изменений и восстанавливать предыдущие версии файлов.

1. **Объясните следующие понятия СКВ и их отношения: хранилище, commit, история, рабочая копия.**

– **Хранилище** – это как архив, где лежат все версии файлов. Оно хранит всё, что происходило с проектом;

– **Commit** – это сохранение изменений. Когда делаешь commit, записываешь состояние проекта в определённый момент времени;

– **История** – это список всех сделанных коммитов. Она помогает увидеть, кто и когда вносил изменения;

– **Рабочая копия** – это та версия проекта, с которой работаешь прямо сейчас.

1. **Что представляют собой и чем отличаются централизованные и децентрализованные?**

– **Централизованные** – это когда есть главный сервер, и все изменения отправляются туда. Все работают через этот сервер;

– **Децентрализованные (например, Git)** – каждый разработчик имеет свою копию проекта, включая всю историю изменений. Можно работать самостоятельно и потом обмениваться изменениями. Это полезно, к примеру, если требуется узнать разные варианты решения задачи от разных разработчиков.

1. **Какие системы контроля версий вы знаете?**

– **Git** — самая популярная сейчас, децентрализованная;

**– BitBucket** – это централизированная система управления версиями.

1. **Какие существуют основные операции в системе контроля версий?**

– **Commit** — сохраняем изменения;

– **Update/Checkout** — получаем последние изменения из общего хранилища;

– **Branching** — создаём отдельную ветку для параллельной работы;

– **Merge** — соединяем изменения из разных веток;

– **Revert** — откатываемся назад, если что-то пошло не так.

1. **СКВ? Приведите примеры СКВ каждого вида.**

– **Централизованные**: BitBucket;

– **Децентрализованные**: Git.

1. **Опишите действия с СКВ при единоличной работе с хранилищем.**
2. Клонируешь проект.
3. Работаешь, вносишь изменения.
4. Делаешь commit.
5. Повторяешь, если надо.
6. **Опишите порядок работы с общим хранилищем в централизованной СКВ.**
7. Клонируешь проект с основного сервера.
8. Работаешь, вносишь изменения.
9. Делаешь commit.
10. Отправка изменений в общее хранилище (push)
11. Повторяешь, если надо.
12. **Что такое и зачем может быть нужна разность (diff)?**

**Разность (diff)** — это процесс сравнения двух версий файлов, позволяющий увидеть изменения, которые были внесены между ними. Это помогает понять, какие изменения были сделаны и почему.

1. **Что такое и зачем может быть нужно слияние (merge)?**

Merge нужен, чтобы соединить изменения из разных веток или версий файла. Например, ты доработал одну фичу, а твой коллега — другую. Теперь надо всё это слить в одну версию.

1. **Что такое конфликты (conflict) и каков процесс их разрешения (resolve)?**

Конфликты возникают, когда два человека меняют одни и те же строки в одном файле. Система не может сама решить, чьё изменение оставить, поэтому тебе приходится вручную выбирать правильное решение.

1. **Поясните процесс синхронизации с общим хранилищем («обновления») в децентрализованной СКВ.**

– **Pull** — получаешь изменения из удалённого репозитория.

– **Push** — отправляешь свои изменения туда.

1. **Что такое и зачем могут быть нужны ветви (branches)?**

Ветви позволяют параллельно работать над разными частями проекта. Например, можно создать новую ветку для добавления новой функции, а в это время другие члены команды продолжают работать над основной версией. Позже можно будет слить изменения из этой ветки в основную.

1. **Объясните смысл действия rebase в СКВ Git.**

Rebase позволяет переписать историю коммитов. Например, можно сделать так, чтобы одни коммиты шли после коммитов другого человека, а не перемешивались с ними.

1. **Как и зачем можно игнорировать некоторые файлы при commit?**

Игнорировать файлы при **commit** можно с помощью файла **.gitignore**. Это нужно, чтобы не путать репозиторий временными файлами или файлами, которые не нужны другим разработчикам. Например, можно игнорировать файлы логов или временные файлы IDE.

1. **Что такое системы контроля версий (VCS) и для решения каких задач они предназначаются?**

**Version Control System (**VCS**)** — это инструменты, которые помогают разработчикам управлять изменениями в файлах проекта. Они позволяют отслеживать изменения, сделанные в коде, сохранять историю изменений и восстанавливать предыдущие версии файлов.

1. Объясните следующие понятия VCS и их отношения: хранилище, commit, история, рабочая копия.

– **Хранилище** – это как архив, где лежат все версии файлов. Оно хранит всё, что происходило с проектом;

– **Commit** – это сохранение изменений. Когда делаешь commit, записываешь состояние проекта в определённый момент времени;

– **История** – это список всех сделанных коммитов. Она помогает увидеть, кто и когда вносил изменения;

– **Рабочая копия** – это та версия проекта, с которой работаешь прямо сейчас.

1. **Что представляют собой и чем отличаются централизованные и децентрализованные VCS? Приведите примеры VCS каждого вида?**

– **Централизованные** – это когда есть главный сервер, и все изменения отправляются туда. Все работают через этот сервер;

– **Децентрализованные (например, Git)** – каждый разработчик имеет свою копию проекта, включая всю историю изменений. Можно работать самостоятельно и потом обмениваться изменениями. Это полезно, к примеру, если требуется узнать разные варианты решения задачи от разных разработчиков.

1. **Опишите действия с VCS при единоличной работе с хранилищем.**
2. Клонируешь проект.
3. Работаешь, вносишь изменения.
4. Делаешь commit.
5. Повторяешь, если надо.
6. **Опишите порядок работы с общим хранилищем VCS.**
7. Клонируешь проект с основного сервера.
8. Работаешь, вносишь изменения.
9. Делаешь commit.
10. Отправка изменений в общее хранилище (push)
11. Повторяешь, если надо.
12. **Каковы основные задачи, решаемые инструментальным средством git?**
13. Управление историей изменений.
14. Поддержание различных ветвей разработки.
15. Удобство совместной работы команды.
16. Возможность восстановления предыдущих версий.
17. **Назовите и дайте краткую характеристику командам git.**

– **git init** – инициализация нового репозитория;

– **git clone** – клонирование существующего репозитория;

– **git** **push** – Команда «git push» используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий. Этой команде нужно право на запись в репозиторий, поэтому она использует аутентификацию;

– **git** **add** – добавление изменений в индекс;

– **git** **commit** – фиксация изменений данных, которые были добавлены в индекс ранее;

– **git** **push** – отправка изменений на удаленный сервер;

– **git** **pull** – Команда «git pull» работает как комбинация команд «git fetch» и «git merge», т. е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой;

– **git** **branch** – создание новых веток;

– **git merge** – слияниеветок.

1. **Приведите примеры использования при работе с локальным и удалённым репозиториями.**

При работе с локальными репозиториями:

– Создаем новый проект: **git** **init**;

– Добавляем файлы: **git** **add…**;

– Фиксируем изменения: **git** **commit** **-m** **"Initial** **commit"**.

При работе с удаленными репозиториями:

– Клонируем проект: **git** **clone** **https://github.com/user/repo.git**;

– Отправляем изменения: **git push origin main**;

– Получаем обновления: **git pull origin main**.

1. **Что такое и зачем могут быть нужны ветви (branches)?**

Ветви позволяют параллельно работать над разными частями проекта. Например, можно создать новую ветку для добавления новой функции, а в это время другие члены команды продолжают работать над основной версией. Позже можно будет слить изменения из этой ветки в основную.

1. **Как и зачем можно игнорировать некоторые файлы при commit?**

Игнорировать файлы при **commit** можно с помощью файла **.gitignore**. Это нужно, чтобы не путать репозиторий временными файлами или файлами, которые не нужны другим разработчикам. Например, можно игнорировать файлы логов или временные файлы IDE.