Это шпаргалка

4.1. Жизненный цикл фронтенд-проекта и основы Git

Привет, я Михаил, преподаватель курса «Инструменты разработки».

Сначала проведу краткий обзор того, чему будет посвящен данный курс. Вы теперь хорошо знакомы с HTML, CSS и умеете использовать JavaScript в браузере. Наконец пришло время познакомиться с большим и важным, скажем так, *джентльменским набором разработчика*.

На этом курсе вы:

- Познакомитесь с жизненным циклом приложения.
- Познакомитесь с понятием «пакет» и узнаете, зачем они нужны в разработке ПО
- Научитесь работать с пакетными менеджерами.
- Освоите систему контроля и управления версиями (Git).
- Познакомитесь с инструментами автоматического форматирования кода.
- Научитесь работать с инструментами автоматической проверки качества кода (линтерами).
- Освоите webpack для сборки приложения.
- Изучите основы TypeScript.

Если ничего из вышеперечисленного не было вам понятно — отлично. Всё это мы поэтапно разберем с вами на занятиях.

А теперь к материалу первого урока!

🎁 **Это может пригодиться!**

Презентация:

https://drive.google.com/file/d/1syorM8CoSCmXyr9G0X_Pktrc2ycmgUZj/view?usp=sharing

Жизненный цикл фронтенд-проекта

С чего же всё начинается? Очевидно, с клетки. Шучу, с идеи.

Любой проект начинается с подготовительной фазы: идея или концепция анализируется, тестируется на жизнеспособность, проводится финансовый анализ или анализ рынка с конкурентами, если они есть. Ведь нет смысла разрабатывать приложение, которое не нужно рынку или просто финансово не выгодно.

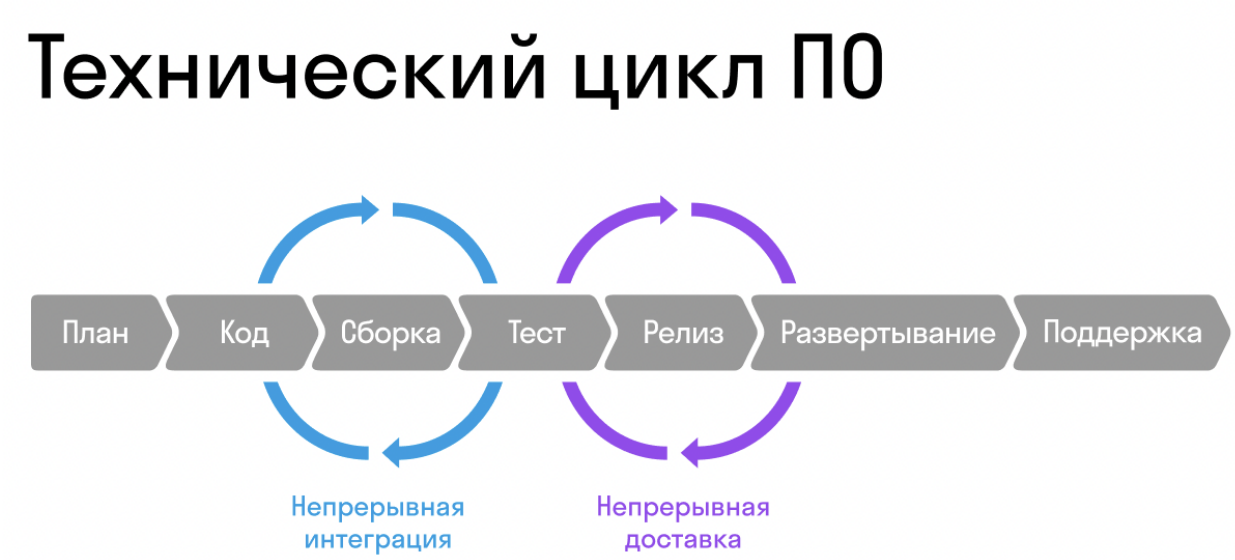
Допустим, мы поняли, что рынок ждет нас! Тогда вполне можно приступить к разработке приложения. Какой процесс происходит?

🔄
1. Идея ->
2. Проверка концепции (бизнес/финансовый анализ) ->
3. Проверка концепции (MVP) ->
4. Разработка ->
5. Поддержка

Технический жизненный цикл приложения

Технический цикл нас интересует больше всего.

Перед вами наглядная схема цикла. Данный цикл называется **CI/CD.**



📌 **CI (Continuous Integration)** — это набор инструментов и практик по автоматизации сборки, линтинга, тестирования и обновления кодовой базы для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

Сборка проекта обычно включает в себя следующее:

- Минификация — это процесс оптимизации кода. Например, из файлов удаляются пробелы или длинные строки меняются на небольшое количество символов. Так как любой символ или пробел — прибавка к весу файла.
- Вырезается весь код, который нужен для разработки. Например, скрипты для сборки в самой сборке не нужны.

📌 **CD (Continuous Deployment)** — набор инструментов и практик по автоматизации доставки обновленного кода в production-среду.

Что касается CD-части процесса, то этим чаще всего занимаются специалисты, которых называют **operation engineer.** Если совсем упрощать, то это роль системного администратора.

Состав продуктовой команды

Как правило, не один человек тащит на себе весь проект.

У вас будет команда (*читайте: братья по несчастью*)!

В сбалансированной команде вы можете встретить следующие роли:

▼ **PO/PM** (project owner / project manager)

- Руководитель проекта, входная точка в вашу команду.

В обязанности входит:

- Общение с заказчиком.
- Получение и обработка требований к продукту от заказчика.
- Расстановка приоритетов для команды.
- Управление командой (распределение времени, задач, целей).

▼ **Аналитик**

В обязанности входит:

- Составление требований к задаче.
- Формулировка и постановка задач.
- Сбор необходимой информации для передачи задачи в разработку.
- Описание документации по функционалу.

▼ **QA-инженер (тестировщик)**

В обязанности входит:

- Проведение ручных тестов.
- Написание автоматических тестов.

▼ **Operation engineer**

В обязанности входит:

- Настройка и поддержка инфраструктуры (серверы, сервисы).
- Раскатка свежего кода на production-среду.
- Написание CI/CD pipelines.

▼ **Team lead frontend-команды**

- Teamlead frontend-команды — *батя* фронтенда :)

В обязанности входит:

- Назначение задач на конкретных исполнителей.
- Выбор технологий и методологий.
- Мониторинг психологического и эмоционального состояния команды.
- Поиск точек роста конкретного разработчика и составление планов роста (KPI и другие методологии).

▼ Frontend-разработчик

В его обязанности входит создание и поддержка фронтенд-приложения (визуальная часть продукта), обработка пользовательских действий в приложении, сбор и отправка данных на бэкенд.

▼ **Team lead backend-команды**

Функции тим.лида бэкенда сильно схожи с тим.лидом фронтенда.

▼ Backend-разработчик

В его обязанности входит создание сервисов и написание основной логики работы приложения. Манипуляции (сохранение, удаление и обновление) данными, которые приходят с веб-приложения, например.

Много, не правда ли? Будем честны, это пример полной продуктовой команды. Вам следует это знать, потому что фронтовое приложение — это лишь часть большого продукта.

Команда собрана. Что дальше?

Вот мы собрали команду. Каким будет следующий этап для команды разработки?

А дальше всё с самого начала:

- Создается репозиторий — место, где будет храниться код, над которым будет работать команда.
- Настраиваются права доступов.
  Права доступов строго ограничены, чтобы младшие разработчики, да и любые посторонние, не внесли хаос в кодовую базу или в жизненный цикл приложения. Например, по неосторожности можно снести базу данных, выкатить в продакшен непроверенный код и так далее.
- Настраиваются проверщики кода (линтеры). Они позволяют ввести четкие правила по стилизации и написанию кода, чтобы кодовая база выглядела консистентно (eslint, prettier).
- Устанавливаются базовые/первичные зависимости (модули). Модули или библиотеки — это готовые (независимые) куски кода, которые решают конкретную задачу. Например, библиотека для слайдера.

Что такое Git?

А теперь к конкретике — поговорим о Git.

Что такое репозитории?

📌 **Git** — распределенная система управления версиями (VCS).

> **VCS** *(version control system)* — это программа для управления и отслеживания изменений в документах, программах и в целом любой информации. В нашем случае — кода.

Что дает нам это управление?

- Возможность посмотреть историю изменений.
- Возможность посмотреть разницу между предыдущим и актуальным состоянием.
- Откатить изменения, которые были внесены 1, 2 или 10 лет назад, а следовательно, посмотреть актуальное на тот момент состояние информации.
- Вносить изменения параллельно (ветвление, branching).

▼ **Git-словарь**

- Репозиторий — папка или место, где находится ваш проект.
- GitHub — сервис, который позволяет хостить (хранить удаленно) ваши репозитории.

▼ **Git-команды**

- `git clone` — клонирование репозитория из удаленного хранилища на локальную машину.
- `git add` — отслеживание изменений в файле/файлах.
- `git commit` — сохранение изменений в Git.
- `git push` — выгрузка сделанных изменений в удаленный репозиторий (например, GitHub).
- `git pull` — получение изменений из удаленного репозитория на локальную машину.
- `git status` — просмотр списка измененных файлов.
- `git stash` — чтобы не создавать коммит, можно поместить изменения во временное хранилище. Это позволит нам сохранить наши изменения и переключаться между ветками без необходимости перед переключением делать коммит.

- `git stash pop` — достать последние изменения из хранилища
- `git remote add origin <url>` — добавление удаленного репозитория в реестр локальных репозиториев.
- `git diff` — просмотр разницы между двумя коммитами.

SSH-ключи

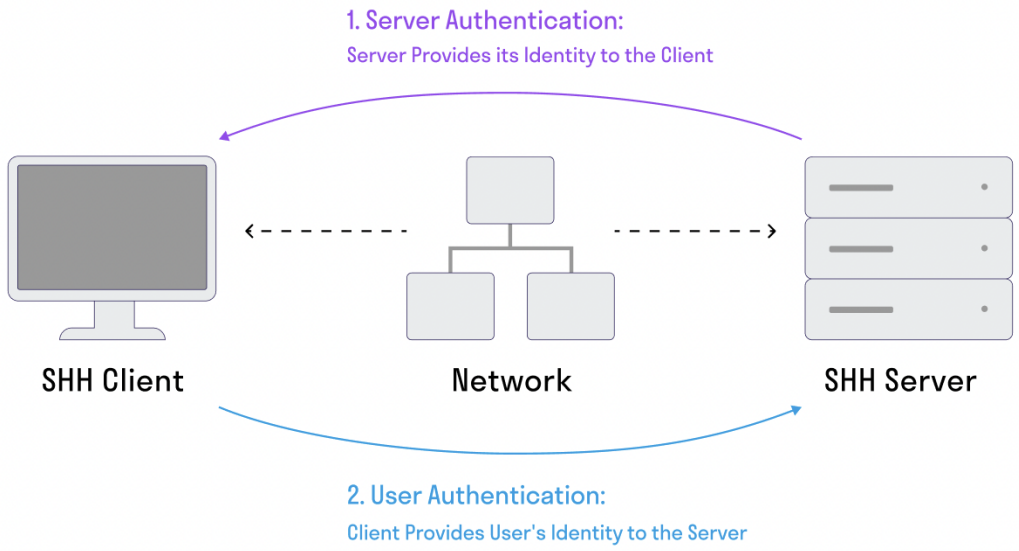
▼ **Что такое SSH-ключи?**

SSH-ключи — это два файла, каждый из которых содержит в себе часть шифра.

Один файл **публичный,** помещается в сервис, с которым мы хотим работать.

Второй файл **приватный**, остается у вас на компьютере. Мы никогда не должны его скомпрометировать, иначе нас обвинят в сломе Белого дома. Шучу, но отчасти. Данный ключ — как ключ от вашей квартиры. Буквально.

При попытке соединения с сервисом ключи сопоставляются. Если всё хорошо, система вас авторизует. Таким образом, мы можем устанавливать соединение с сервисом.



▼ **Как настроить SHH-ключи**

Прежде чем приступить к практическим примерам, мы должны настроить доступы, чтобы можно было взаимодействовать с удаленным репозиторием. А если быть точнее, мы должны настроить доступы через SSH-ключи.

Инструкция по настройке SSH keys:
https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

▼ **Генерация ключей**

Окей, давайте разберем на практике.

```
# Генерируем ключ
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"

# Запускаем ssh-agent
eval "$(ssh-agent -s)"

# Открываем ssh-конфиг файл
open ~/.ssh/config

# Добавляем ключ в файл
Host *
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/<FILE-NAME>

# Добавляем приватный ключ в ssh-agent
ssh-add -K ~/.ssh/FILE-NAME

# Копируем публичный ключ в буфер обмена
pbcopy < ~/.ssh/<FILE-NAME>.pub

# Идем в сервис и добавляем ключ
```

Commit

> **Commit** — это слепок всей информации в репозитории в определенное время.

Commit содержит информацию:

- Название.
- Описание (опционально).
- Дата создания.
- Информации о создателе.
- Уникальный идентификатор.

Обратите внимание, что commit — это полноценный слепок репозитория в момент времени, а не сами изменения. Git вычисляет изменения из двух слепков. Изменение между слепками (commits) называется diff (от английского difference).

Практика. Создадим репозиторий локально

```
# Локально заинициализируем репозиторий
git init

# Создадим в GitHub чистый репозиторий

# Свяжем локальный и удаленный репозитории
git remote add origin <URL>

# Отправим данные из локального репозитория в удаленный
git push origin master
```

Branching

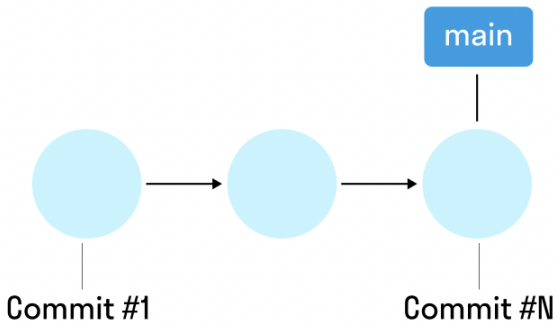
Поговорим теперь о ветвлении истории изменений.

▼ **Словарь и подробности**

- `Branch` — ветка. Отдельная версия приложения.
- `Master, main` — главная ветка.
- `Feature branch` — ветка, на которую вносятся какие-то новые изменения в код. Новый функционал, к примеру.
- `Fix branch` — ветка, в которой вносятся исправления в код.

Ранее мы с вами посмотрели, как работает Git. Но мы вносили все изменения в корневую версию проекта. Эта версия проекта называется master-ветка (она же main-ветка).

Разберем на схеме.



Ранее мы работали по такой схеме, она линейна. Лучшая аналогия — течение времени или хронологическая линия.

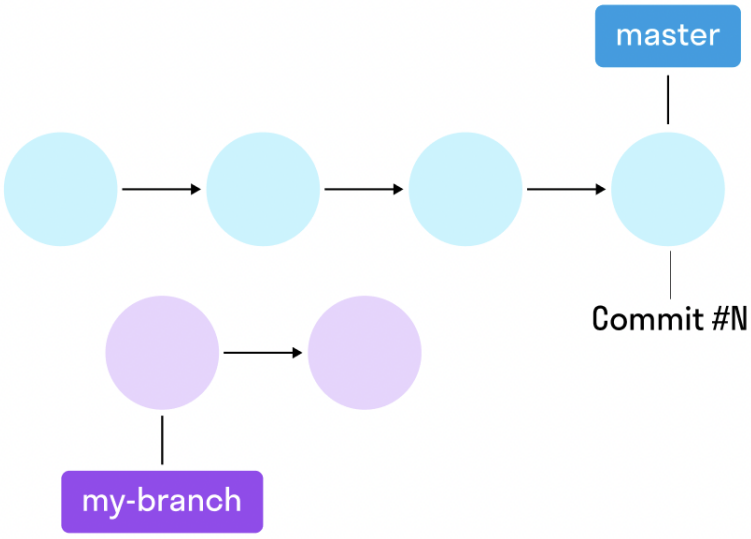
Каждое звено в этой «хронологической цепи» — **commit**. А как мы помним, commit — полноценный слепок информации в определенный период времени. Но данную историю можно разветвить. Зачем это нужно?

Например, это позволяет разрабатывать одну программу большому числу

людей одновременно. Или одному разработчику переключаться между разными контекстами (разными задачами, не зависящими друг от друга).

По сути, **branch** (ветка) — это отдельное направление разработки.

Давайте рассмотрим схематично на ветке.



Допустим, есть репозиторий с 2 ветками (master и my-branch). На схеме видно, что есть основная ветка master, а сбоку есть другая ветка — my-branch.

Во второй ветке могут вноситься абсолютно разные изменения, и они никаким случайным образом не затронут основную ветку.

▼ Branching. Практика

Итак, мы на **master-ветке** в нашем репозитории. Чтобы создать ветку и сразу же на нее переключиться, воспользуемся командой:

```
# Создать ветку и сразу же на нее переключиться
git checkout -b "branch-name"
```

Параметр **-b** означает, что мы хотим создать новую ветку. Забегая вперед,

скажем, что с помощью команды **git checkout** можно переключаться между уже существующими ветками:

```
# Переключиться на существующую ветку
git checkout "branch-name"
```

Если вы хотите просто создать ветку, без переключения на нее, воспользуйтесь командой:

```
# Переключиться на существующую ветку
git branch "branch-name"
```

Вот мы на ветке. Давайте посмотрим, сколько у нас вообще есть веток в репозитории:

```
# Вывести список всех веток в репозитории
git branch --list
```

Также мы можем переименовать нашу ветку, если вдруг пожелаем:

```
# Переименовать ветку
git branch -m "new-branch-name"
```

▼ Отправка локальной ветки в удаленный репозиторий

Теперь давайте перенесем нашу локальную ветку в удаленный репозиторий. *К примеру, чтобы наш коллега мог убедиться, что мы весь день не в потолок плевали*.

```
git push origin "branch-name"
```

Аргумент **origin** говорит нам, что мы хотим отправить ветку "branch-name" из локального репозитория в удаленный.

Отлично, идем в GitHub и проверим.

Merge request / Pull Request

Теперь рассмотрим концепцию **merge request** (или pull request, всё зависит от терминологии инструмента).

Если переводить дословно, то это означает запрос на слияние.

> **Pull request —** запрос на слияние кодовой базы из одной ветки в другую ветку.

В разработке мы не вносим напрямую изменения в **master**, это строго запрещено. Перед тем, как наш код попадет в общую кодовую базу, он должен пройти так называемое **review** и тестирование.

Review — проверка качества кода нашими коллегами. В **PR** можно оставлять комментарии, вести дискуссии. Когда изменения одобрены коллегами, а код протестирован, то изменения можно вливать.

Обновление локальной кодовой базы

А теперь мы оказались в ситуации, когда **удаленный** репозиторий более актуален (сточки зрения кодовой базы), чем наш **локальный** репозиторий. Не беда.

Актуальный код лежит в удаленном репозитории в **master-ветке**, но в нашей локальной **master-ветке** код старый. Исправляем!

```
# Переключаемся на master
git checkout master

# Подтягиваем код из удаленного репозитория в локальный
git pull origin master
```

Видите аргумент **origin**? Знакомо? По факту, мы запросили свежую версию из удаленного репозитория в наш локальный, и теперь у нас свежая версия.

Резюме урока

- Познакомились с философией Git (коммиты, ветки).
- Освоили базовые команды Git.
- Настроили SSH-ключи.
- Создали репозиторий через GitHub.
- Создали репозиторий локально с помощью Git.