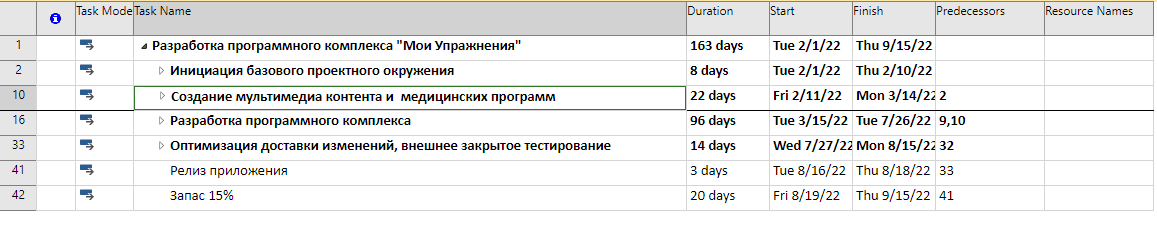
How to BS Design - MyExercise

ООО “БС Дизайн” - мололодая компания, основанная в 2021 году в г. Саратов. Основной вид деятельности компании - разработка программного обеспечения, связанного с посттравматическим и послеоперационным восстановлением пациентов

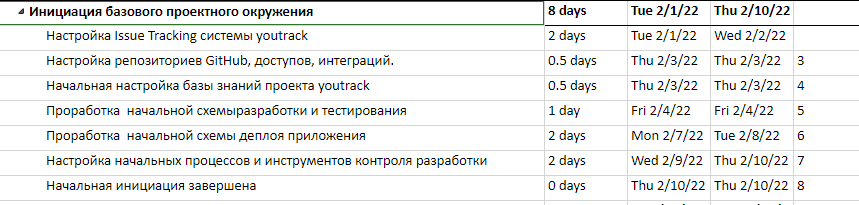
# О проекте

* Целью проекта является создание универсального ядра программного комплекса “Мои упражнения” и двух курсов: “Восстановление ПКС” и “Предотвращение травм ПКС”\*
* Сервис “Мои упражнения” служит для помощи в восстановления после травм опорно-двигательного аппарата за счет алгоритмической индивидуализации комплексов упражнений на основе особенностей проведенной операции, типа травмы, вида спортивных занятий и т.п.
* Для клиник, врачей и НИИ данный продукт представляет возможность получать обезличенную статистику прогресса восстановления пациентов после травм, а также участвовать в коммерческих партнерских программах.
* В перспективе сервис “Мои упражнения” должен стать платформой для независимых экспертов в области восстановительной медицины, которые смогут размещать на ней свои курсы восстановления; аналитическое ядро системы на основе статистики позволит непрерывно улучшать качество восстановительных программ.
* Программный комплекс “Мои упражнения” в разработанной конфигурации должен поддерживать 10000 активных пользователей системы.

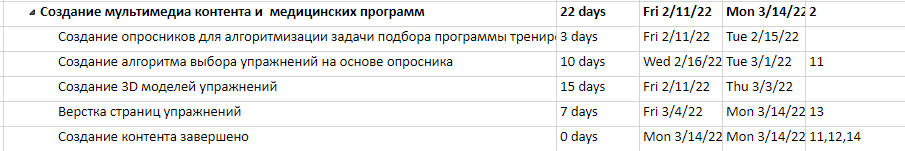
# Этапы проекта



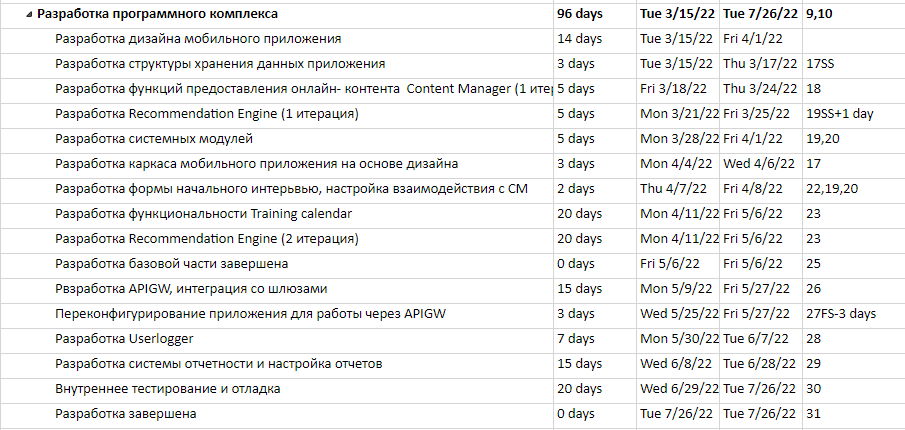
## Инициация базового проектного окружения



## Создание мультимедиа контента и медицинских программ



## Разработка программного комплекса



## Оптимизация доставки изменений, внешнее закрытое тестирование



## Роли и структура команды

| Роль | Ф.И.О | Обращаться по вопросам |
| --- | --- | --- |
| Основатели\владельцы | Бычков Александр  Шабинский Сергей | Стратегические вопросы |
| Генеральный директор | Бычков Александр | Сотрудничество  Финансовые вопросы  Маркетинг и продвижение  Дизайн и функциональность решения |
| Технический директор | Шабинский Сергей | Архитектура решения  Дизайн и функциональность решения  Процессы организации разработки, внедрения и поддержки  Доступ к информационным ресурсам (youtrack, база знаний, github)  Стенды разработки и тестирования  DevOps |
| Консультант по мед вопросам. | Корнилов |  |
| Управление проектом | Бычков Александр | План и скоуп проекта |

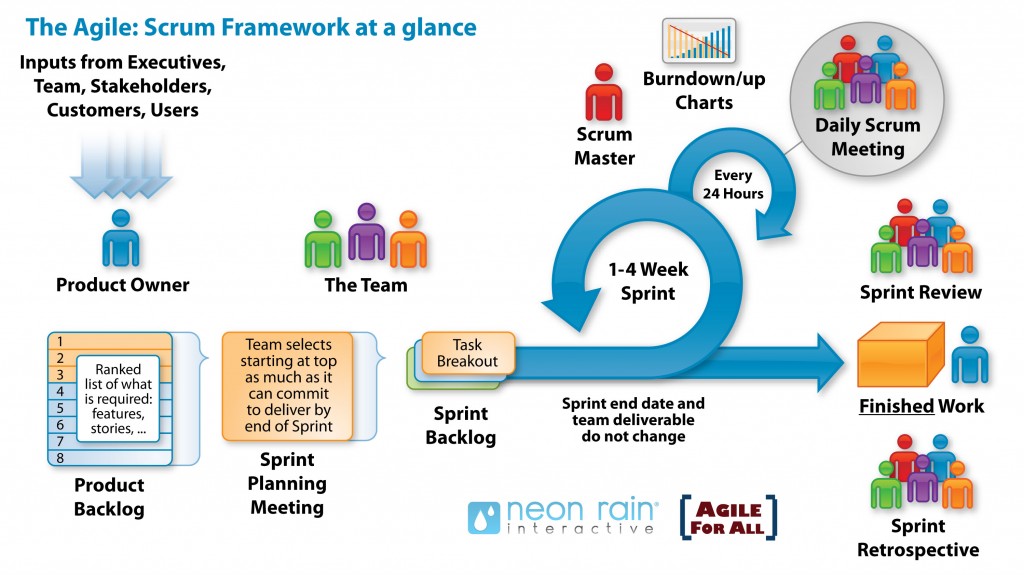
# Правила и инструменты работы

## Управление проектом

Разработка инновационного продукта связана с довольно большой неопределенностью в техническом задании и представлении о конечном варианте продукта. Из этого следует необходимость использования итерационных методов управления проектом (итеративный + инкрементальный).

Управление проекта будет осуществляться по идеям SCRUM. Напомним основные идеи

## Процесс Scrum



Основой Scrum является Sprint, в течении которого выполняется работа над продуктом. По окончанию Sprint должна быть получена новая рабочая версия продукта. Sprint имеет продолжительность, равную 1 неделе и имеет одинаковую продолжительность на протяжении всей жизни продукта.

Перед началом каждого Sprint производится Sprint Planning, на котором производится оценка содержимого Product Backlog и формирование Sprint Backlog, который содержит задачи (Story, Bugs, Tasks), которые должны быть выполнены в текущем спринте.

Каждый день производится Daily Scrum, на котором каждый член команды отвечает на вопросы

1. «что я сделал вчера?»,
2. «что я планирую сделать сегодня?»
3. «какие препятствия на своей работе я встретил?»

Задача Daily Scrum — определение статуса и прогресса работы над Sprint, раннее обнаружение возникших препятствий, выработка решений по изменению стратегии, необходимых для достижения целей Sprint'а.

По окончанию Sprint'а производятся Sprint Review и Sprint Retrospective, задача которых оценить эффективность (производительность) команды в прошедшем Sprint'е, спрогнозировать ожидаемую эффективность (производительность) в следующем спринте, выявлении имеющихся проблем, оценки вероятности завершения всех необходимых работ по продукту и другое.

## Роли SCRUM

В классическом Scrum существует 3 базовых роли:

-*Product owner*

-*Scrum master*

-*Команда разработки (Development team)*

*Product owner* (PO) является связующим звеном между командой разработки и заказчиком. Задача PO — максимальное увеличение ценности разрабатываемого продукта и работы команды.

Одним из основных инструментов PO является Product Backlog. Product Backlog содержит необходимые для выполнения рабочие задачи (такие как Story, Bug, Task и др.), отсортированные в порядке приоритета (срочности).

*Scrum master* (SM) является «служащим лидером» (англ. servant-leader). Задача Scrum Master — помочь команде максимизировать ее эффективность посредством устранения препятствий, помощи, обучении и мотивации команде, помощи PO

*Команда разработки* (Development team, DT) состоит из специалистов, производящих непосредственную работу над производимым продуктом.

Согласно The Scrum Guide (документу, являющимся официальным описанием Scrum от его авторов), DT должны обладать следующими качествами и характеристиками:

-Быть самоорганизующейся. Никто (включая SM и PO) не может указывать команде, как им преобразовать Product Backlog в работающий продукт

-Быть многофункциональной, обладать всеми необходимыми навыками для выпуска работающего продукта

-За выполняемую работу отвечает вся команда, а не индивидуальные члены команды

## Роли BS Дизайн в разрезе управления проектом +RACI

| **Роль Scrum** | **Роль в команде** | **Responsible** | **Accountable** | **Consult** | **Informed** |
| --- | --- | --- | --- | --- | --- |
| Команда разработки | Разработка архитектуры | Шабинский | Шабинский | Программист 1 | Бычков |
|  | Разработка интерфейсов |  |  |  |  |
|  | Маркетинг-view |  |  |  |  |
| *Scrum master* |  |  |  |  |  |
| *Product owner* |  |  |  |  |  |

## Инструменты

Проекты Youtrack, обеспечивающие деятельность BS-Design

* BSTASKS- Общие задачи компании, не связанные непосредственно с разработкой ПО.
* BSDEV - Задачи компании ,связанные с разработкой ПО

## Проект BSDEV

| **Статусы** | **Описание** |
| --- | --- |
| OPEN | Начальный статус при создании задачи. |
| PREPARING | Подготовка к выполнению |
| IMPLEMENTATION | Реализация задачи |
| VALIDATING | Тестирование |
| VALIDATED | Задача успешно протестирована |
| POSTPONED | Задача приостановлена |
| DEPLOYING | Производится установка |
| CLOSED | Закрыта |
| DEPRECATED | Задача устарела |

* *показаны только основные переходы*

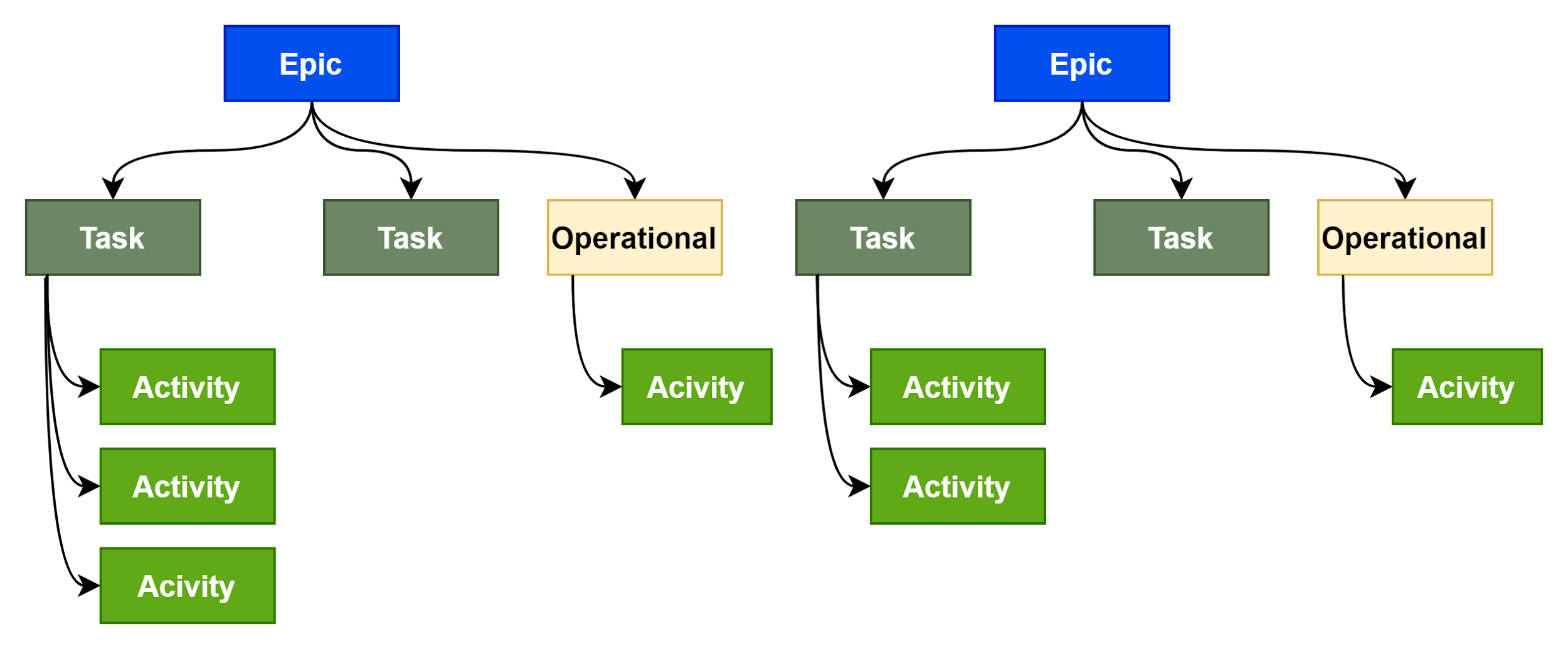
| Шаг | Исполнитель | Система | Описание |
| --- | --- | --- | --- |
| 1.Взятие задачи в работу | Программист | Youtrack | Исполнитель берет задачу в работу. Постановка на задачу может находится в поле Description, приложенном файле либо в прилинкованной задаче SyRS. |
| 2. Подготовка к выполнению задачи. | Программист  ИТ-инженер |  | Исполнитель переводит задачу в статус PREPARING  Исполнитель уточняет необходимую информацию по задаче., а также запрашивает необходимые ресурсы для выполнения задачи |
| 2. Разработка ПО | Программист | Youtrack  GitHub  Yandex Cloud | Проводит разработку ПО согласно инструкциям по разработке ПО (сделать документ)  По готовности к тестированию переводит тикет в статус VALIDATION  и переводит тикет на тестировщика |
| 3. Тестирование ПО | Тестировщик |  | Исполнитель проводит тестирование . Если найдены ошибки, переводит тикет в статус DEVELOPING с указанием причины в ноте к задаче  При успешном завершении тестирования переводит в статус VALIDATED |
| 1. Установка | Установщик |  | Исполнитель проводит установку согласно правилам установки |
| 1. Закрытие задачи | Установщик | Youtrack | После установки производится закрытие задачи |

## Роли

| Роль | Описание |
| --- | --- |
| Программист | ведет разработку кода продукта |
| Тестировщик | Проводит тестирование продукта |
| Установщик | Проводит установку на пром. среду. |
| Инженер-ИТ | Обеспечивает необходимую ИТ-инфраструктуру для разработки и внедрения |

## Проект BSTASKS

### Task hierarchy model

****

### Status flow

# 

| **Issue type** | **Additional attribute** | **Description** |
| --- | --- | --- |
| Goal | Goal type = Company goal | Company goal.  Responsible: \_\_\_  **Achievement results:**  1. achieved beyond expectations  2. achieved according expectation  3. achieved with minor issues  4. achieved with significant issues  not achieved  Workflow  Open, in progress, resolved, reopned,closed |
| GoalTask | - | Atomic Task directly connected to Goal |
| Goal | Goal type = Unit goal | Cascaded from Company goal by issue link =  \* Major effect  \* Medium effect  \* Minor effect |
| Epic | Epic type = Project | Project. May be linked to some Goals |
| Epic | Epic type = Operations | Operational work. |
| Task | - | one-time task, connected to Epic with epic link |
| Sub-task | - | sub-task for task. For decomposition of complicated tasks |
| Function | - | Regular operational task for one role |

### Порядок работы и инструменты SCRUM

| Инструмент |  |
| --- | --- |
| Issue проекта BSDEV | <https://enproxima.myjetbrains.com/youtrack/issues?q=project:%20BSDEV%20> |
| SCRUM доска | <https://enproxima.myjetbrains.com/youtrack/agiles/120-9/current> |
| Burndown диаграмма | <https://enproxima.myjetbrains.com/youtrack/agiles/120-9/current?chart> |

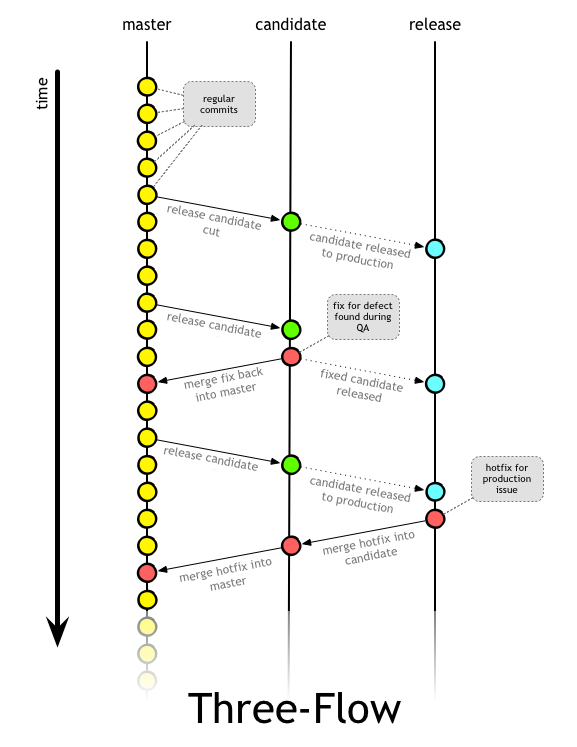
## Коммуникации в проекте

| **Активность** | **Цель** | **Начало и продолжительность** | **Участники** | **Канал взаимодействия** |
| --- | --- | --- | --- | --- |
| Ежедневный SCRUM |  | 9:00 15 минут |  | Zoom |
| Ретроспектива Sprinta |  | каждые 2 недели, суббота 9-00, 1 час |  | Zoom |
| Встреча WSM (weekly status meeting) |  | 1 час, либо 2 раза по 3 0минут | Бычков  Шабинский | Zoom |
| Срочные вопросы по технике |  |  | Основной:  Резервный: |  |
| Вопросы по управлению |  |  |  |  |
| Коммуникация по новостям, сервисам и т.п |  |  |  | Канал в slack |

## 

## Процесс разработки

# Схема ветвления git

В проекте используется схема ветвления [ThreeFlow](https://habr.com/ru/company/infopulse/blog/345826/) 

Все разработчики работают в одной мастер-ветке. Если разрабатываемая фича тривиальна — она просто реализуется и добавляется одним комитом. Если разработка фичи займёт какое-то время, то сначала добавляется флаг (по-умолчанию выключенный) для её активизации (см. работа со флагами)

## Релизы

Когда приходит время релиза делается «срез» ветки master в ветку релиз-кандидатов. Одна и та же ветка используется для всех релиз-кандидатов.

Предназначение данной ветки — дать билд, который получит команда QA для выполнения тестов.

Для создания релиз-кандидата вы делаете что-то вроде этого:

$ git checkout candidate # предполагаем, что candidate указывает на origin/candidate

$ git pull # убедимся, что у нас актуальная копия репозитория

$ git merge --no-ff origin/master

$ git tag candidate-3.2.645

$ git push --follow-tags

Причина использования флага "--no-ff" здесь в том, что мы хотим создать merge-коммит (новый коммит с двумя родителями). Одним из его родителей будет предыдущий HEAD ветки релиз-кандидатов, а второй — HEAD ветки master. Это позволит вам легко отслеживать в истории кто и когда создал релиз-кандидат, а также что конкретно в него вошло (какие коммиты ветки master).

Также вы могли заметить, что мы создали тег для релиз-кандидата. Чуть детальнее об этом.

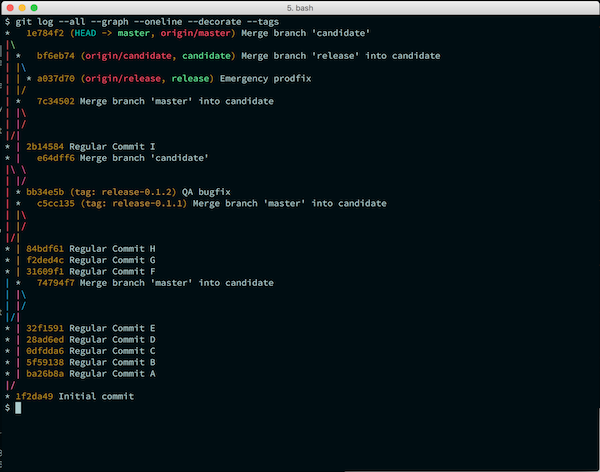
Если при тестировании релиз-кандидата обнаружатся баги, то они будут исправлены прямо в ветке релиз-кандидата, там же будет промаркирован новый релиз-кандидат, а изменения с исправлениями будут смерджены обратно в master. Эти изменения тоже должны быть применены с параметром "--no-ff", ведь мы хотим аккуратно показать, какой именно код был перемещен между ветками.

Когда релиз-кандидат протестирован и одобрен, мы обновляем ветку релизов таким образом, чтобы её HEAD указывал на HEAD ветки релиз-кандидатов. Поскольку у нас есть тег для каждого релиз-кандидата, то мы можем именно его запушить в ветку релизов:

$ git push --force origin candidate-3.2.647:release

Параметр "--force" здесь означает, что мы игнорируем все изменения в ветке релизов и просто насильно устанавливаем её HEAD на тот же коммит, который обозначает последний созданный тег релиз-кандидата (candidate-3.2.647 в примере выше). Заметьте, что это вовсе не является слиянием (merge), но это потому, что нам оно здесь и не нужно. Мы не хотим усложнять историю в Git, да и вообще единственной причиной создания ветки релизов является теоретическая необходимость экстренного фикса обнаруженной на продакшене критической проблемы. Да, этот "--force" перетрёт все хотфиксы в ветке релизов. Но знаете, если вы релизите следующую версию продукта с новыми фичами в то же время, пока другой член вашей команды фиксит баги на продакшене — у вас серьёзные проблемы с управлением проектом и коммуникациями. Их стоит решить ещё до начала всех этих танцев вокруг веток и релизов. Фиксы в ветке релизов должны быть очень редкими и, конечно, должны быть потом смерджены в ветки релиз-кандидатов и master.

Причина, по которой мы используем "--force", а не merge в том, что при merge коммит в HEAD ветки релиз-кандидатов и коммит в HEAD ветки релизов могут иметь разные sha-1, а это не то, что нам нужно. Мы не хотим создавать новый коммит с релизом, мы хотим назвать релизом именно тот коммит, который был выбран релиз-кандидатом, который тестировался командой QA и был одобрен к релизу тем, кто за это отвечает. Именно это и делает "--force".



Если вы будете следовать этим рекомендациям, то история в вашем git-репозитории будет выглядеть очень похоже на рисунок выше, показывая в точности какие коммиты перемещались между ветками.

## Release Notes

Вы можете легко генерировать «release notes» к новым релизам. Нужно всего лишь получить разницу между прошлым тегом релиза и текущим тегом релиз-кандидата. Поскольку в ветке релизов у нас лежит то, что когда-то точно было релиз-кандидатом, мы можем узнать каким именно:

$ git describe --tags release

candidate-3.1.248

Теперь, когда мы знаем, что в релиз-кандидате у нас сейчас candidate-3.2.259, можно получить разницу между этими двумя тегами:

$ git log --oneline candidate-3.1.248..candidate-3.2.259

Ну или даже проще, без тегов, просто сравниваем HEAD веток release и candidate:

$ git log --oneline release..candidate

## Применяемые операции

Вот некоторые часто используемые операции при работе по ThreeFlow. Все примеры предполагают, что ваши локальные ветки правильно соотнесены с удалёнными ветками и содержат актуальные изменения. Если вы не уверены в этом — всегда будет хорошей идеей лишний раз сделать git fetch и потом использовать имена вроде origin/master вместо просто master

Как мне сделать релиз-кандидат из ветки master?

$ git checkout candidate

$ git pull

$ git merge --no-ff master

$ git tag candidate-3.2.645 #optionally tag the candidate

$ git push --follow-tags

Как мне сделать релиз из релиз-кандидата?

$ git push --force origin <tag **for** the candidate>:release

Если вы почему-то решили не тегировать релиз-кандидаты, то придётся сделать:

$ git push --force origin candidate:release

Как мне найти ветку, в которой есть некоторый конкретный коммит?

Иногда люди хотят убедиться, что какое-то конкретное изменение вошло в релиз-кандидат или в релиз. Вот как это проверить:

$ git branch -r -contains <sha of commit>

Как мне найти тег, на который указывает HEAD некоторой ветки?

$ git describe --tags <branch>

Как мне узнать, какие коммиты войдут в некоторый новый релиз?

$ git log --oneline release..<tag of release candidate>

или:

$ git log --oneline release..origin/candidate

Как мне настроить ветки релиз-кандидатов и релизов?

Любой проект начинается с первого коммита. Обычно это что-то простое, вроде добавления readme-файла. Я советую просто сделать ветки релиз-кандидатов и релизов из этого коммита. Что нам нужно получить, это первый merge-коммит с двумя родителями. Таким образом мы получим корректную историю. Так что подойдёт, в общем, любой коммит master-ветки. Почему бы не взять первый?

$ git branch candidate `git log --format=%H --reverse | head -1`

$ git checkout candidate

$ git push

Чтобы сделать ветку для релизов:

$ git branch release

$ git branch release --set-upstream-to=origin/release

## Работа со флагами

## Использование флагов для включения фич

Вместо использования веток для разработки фич, попробуйте [использовать флаги](https://www.martinfowler.com/articles/feature-toggles.html) для их включения-выключения. Это просто. Начните разработку новой фичи с объявления булевого флага, по которому она будет включаться. Установите его по-умолчанию в false — и в этом случае вызывайте старый код, без кода для новой фичи:

**if**(newCodeEnabled) {

// новый код

} **else** {

// старый код

}

Сам флаг может быть как жестко зашит в код, так и вынесен во внешний конфиг (возможно с использованием чего-то типа [Consul](https://www.consul.io/docs/agent/options.html) или [Zookeeper](https://zookeeper.apache.org/)), что даст возможность включать и выключать новую функциональность для тестирования или даже в продакшене.

Когда два разработчика работают в одной (основной) ветке над разными фичами, то создают по флагу на каждую из них. И просто комитят\забирают код регулярно. Шансы на возникновение конфликта в таком случае минимальны. Каждый может комитнуть код, когда считает нужным. Каждый может синхронизировать свой локальный репозиторий с основным — и рассинхрон будет минимальным (уж точно не больше одного рабочего дня). Конфликтов либо не будет вовсе, либо они будут минимальны. Значительно проще понять, что изменил твой коллега вот в этом десятке строк за последний час, чем разгребать глобальные изменения за дни или недели, как предлагает нам GitFlow.

И да, если вы пишете тесты для своего кода , то нужно тестировать и ветку кода с отключенным флагом, и ту, где флаг включен. Если ведётся разработка двух взаимозависимых фич — на время разработки вам понадобится 4 теста для всех их комбинаций. Это звучит как угроза усложнения и замедления разработки, но не забывайте, что после окончания разработки новых фич «старые» блоки кода (и тесты для них) будут удалены, так что геометрического увеличения сложности вы не получите.

Флаги для новых фич могут использоваться и более динамично. Вы можете привязать их к определённым группам пользователей для бета-тестирования или A/B тестов.

Когда разработка фичи завершена и она включена по-умолчанию в продакшене, вы можете запланировать небольшую низкоприоритетную задачу по удалению старого кода и самого флага. Или, если вы по каким-то причинам хотите сохранить возможность отключения фичи (проблемы со стабильностью нового кода, регулирование нагрузки на backend), этого можно и не делать. В любом случае, важно осознанно принять решение об удалении или оставлении флага и старого кода — если об этом забыть, то со временем ваш код обрастёт мхом старого неиспользуемого функционала, который будет только отвлекать разработчиков и не приносить никакой реальной пользы.

Ценность подхода с флагами для включения новых фич просто невозможно переоценить. Я гарантирую, что как только вы начнёте использовать флаги для фич вместо веток для них же, то никогда не захотите вернуться обратно. На моей памяти почти всегда разработка большой новой фичи в отдельной долгоживущей ветке рано или поздно приводила к проблемам, требующим внимания сразу нескольких программистов и глубокого знания Git. В то же время подход с работой в одной ветке и флагами для новых фич ни разу не привёл к каким-то конфликтам, которые нельзя было бы решить за пару минут одним человеком.

## Настройки GitHub

Сканирование кода [Setting up code scanning for a repository](https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/setting-up-code-scanning-for-a-repository)

# Версионирование

Используем семантическое версионирование Подробно тут: <https://semver.org/lang/ru/>

**Кратко**

Учитывая номер версии МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ, следует увеличивать:

1. МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.
2. МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.
3. ПАТЧ-версию, когда вы делаете обратно совместимые исправления.

Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ форма

**FAQ**

### **Что я должен делать с ревизиями в 0.y.z на начальной стадии разработки?**

# Самое простое — начать разработку с 0.1.0 и затем увеличивать минорную версию для каждого последующего релиза.

### **Как я узнаю, когда пора делать релиз 1.0.0?**

# Если ваше ПО используется на продакшене, оно, вероятно, уже должно быть версии 1.0.0. Если у вас стабильный API, от которого зависят пользователи, версия должна быть 1.0.0. Если вы беспокоитесь за обратную совместимость, вероятно, версия вашего ПО уже 1.0.0.

### **Не препятствует ли это быстрой разработке и коротким итерациям?**

# Мажорная версия 0 как раз и означает быструю разработку. Если вы изменяете API каждый день, вы должны быть на версии 0.y.z или на отдельной ветке разработки работать над следующей главной версией.

### **Даже если малейшие обратно несовместимые изменения в публичном API требуют выпуска новой главной версии, не закончится ли это тем, что очень скоро версия станет 42.0.0?**

# Это вопрос ответственной разработки и предвидения. Несовместимые изменения не должны быть представлены как незначительные в ПО, имеющем много зависимого кода. Стоимость обновления может быть велика. Практика увеличения главных версий релизов с обратно несовместимыми изменениями означает, что вам придётся думать о последствиях ваших изменений и учитывать соотношение цена/качество.

### **Документирование всего API — слишком много работы!**

# Это ваша ответственность, как профессионального разработчика, правильно документировать ПО, предназначенное для широкого использования. Управление сложностью ПО очень важная часть поддержки высокой эффективности проекта. Это тяжело сделать, если никто не знает, как использовать ваше ПО или какой метод можно вызывать безопасно. В долгосрочной перспективе Семантическое Версионирование и настойчивость в качественном документировании публичного API поможет всем и всему работать слаженно.

### **Что мне делать, если я случайно зарелизил обратно несовместимые изменения как минорную версию?**

# Как только вы поняли, что нарушили спецификации Семантического Версионирования, исправьте проблему и выпустите новую минорную версию, которая исправляет проблему и восстанавливает обратную совместимость. Даже в таких обстоятельствах неприемлемо модифицировать уже выпущенные релизы. Если это необходимо, укажите в документации о нарушении обратной совместимости, версионирования и проинформируйте ваших пользователей, чтобы они знали о нарушении порядка версий.

### **Что я должен делать, если я обновляю свои собственные зависимости без изменения публичного API?**

# Это можно рассматривать как совместимые изменения, так как они не влияют на публичный API. ПО, которое явно зависит от тех же зависимостей что и ваш пакет, должно иметь собственные спецификации зависимостей и автор будет уведомлен о возможных конфликтах. Являются ли данные изменения уровня патча или минорного уровня, зависит от того, обновили ли вы свои зависимости чтобы исправить баг или реализовать новую функциональность. В последнем случае, как правило, добавляется некоторое количество дополнительного кода и как следствие, увеличивается минорная версия.

### **Что если я нечаянно изменил публичный API в несоответствии с изменением номера версии (т.е. код содержит обратно несовместимые изменения в патч-релизе)?**

# На ваше усмотрение. Если у вас огромная аудитория, которая будет поставлена перед фактом возвращения прежнего поведения API, то лучше выпустить новый релиз с увеличением главной версии, даже несмотря на то, что фикс содержит исправления уровня патча. Запомните, в Семантическом Версионировании номера версий изменяются строго следуя спецификации. Если эти изменения важны для ваших пользователей, используйте номер версии, чтобы информировать их.

### **Что делать с устаревшей функциональностью?**

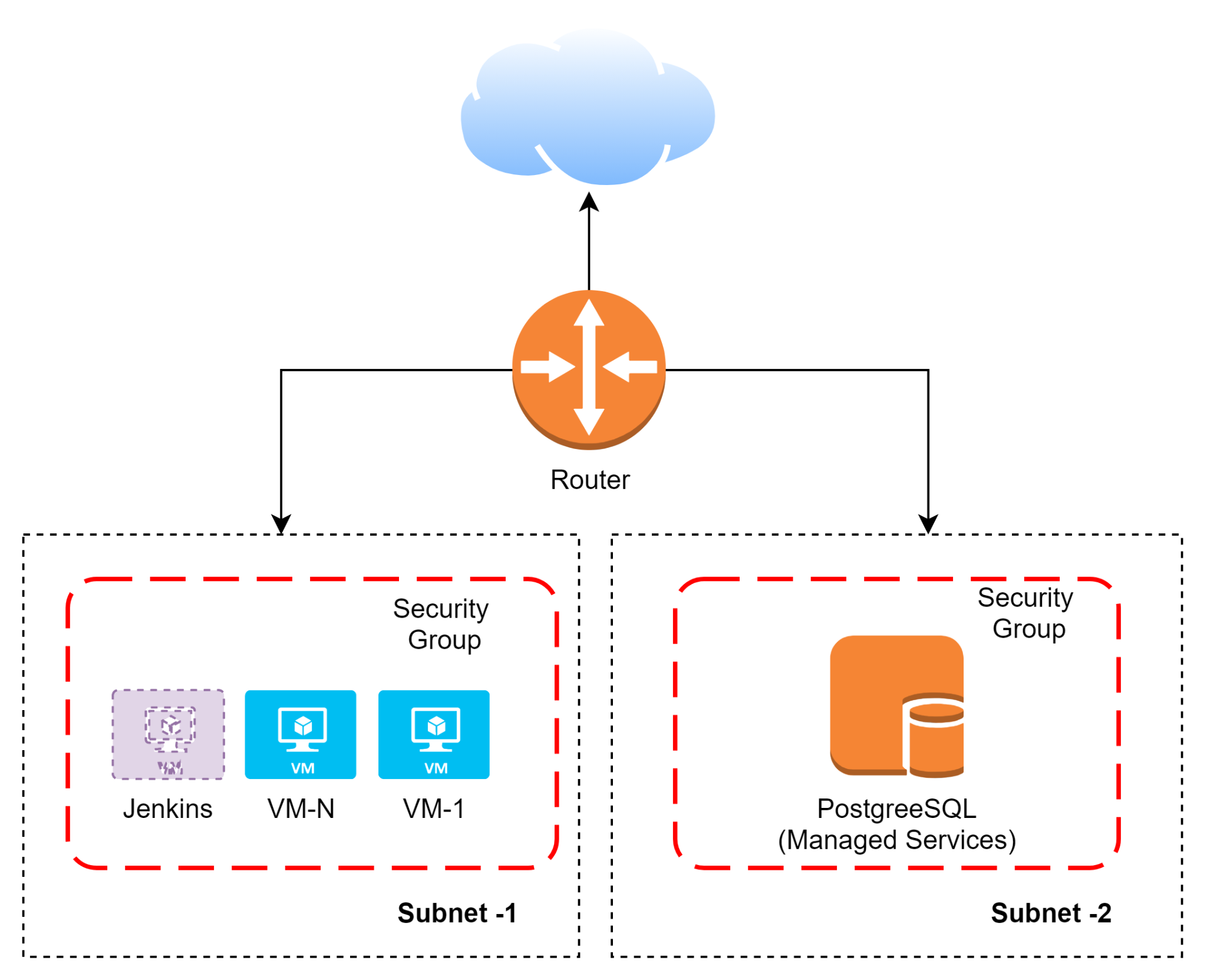
# Объявление функциональности устаревшей — это обычное дело в ходе разработки и часто необходимо для продвижения вперёд. Когда вы объявляете устаревшим часть публичного API, вы должны сделать две вещи: (1) обновить вашу документацию, чтобы дать пользователям узнать об этом изменении; (2) выпустить новый релиз с увеличением минорной версии. Прежде чем вы полностью удалите устаревшую функциональность в релизе с увеличением главной версии, должен быть как минимум один минорный релиз, содержащий объявление функциональности устаревшей, чтобы пользователи могли плавно перейти на новый API.

# Среда разработки

В качестве среды разработки и тестирования предполагается использовать Cloud Mail

ru.

На данный момент проработана следующая архитектура



Имеется возможность разворачивать ее из Terraform. Ссылка на GitHub <https://github.com/bsdesigncompany/VKCloud-IaC>

Разработка ведется локально, тестовый стенд создается в облаке Yandex Cloud.

Реквизиты стенда

| **Сервер** | **IP** | **Назначение** | **Параметры доступа** | **Комментарии** |
| --- | --- | --- | --- | --- |
| Jenkins | будет определен позднее | Выполнение различных скриптов (вкл, выкл серверов, CICD) |  |  |
| VM-1 разработки | будет определен позднее |  |  |  |
| DB |  |  |  |  |

### Правила работы с тестовым стендом

Тестовый стенд должен работать только когда это требуется. В ночное время его следует выключать.

Выключение производится следующим образом:

Jenkins

Включение

:

## СI\CD

CI\CD будет развиваться постепенно, по мере развития проекта.

Доступные фичи:

1. Jenkins: Копирование из репозитория GIT на сервер
2. Ansible: ad-hoc команды

## База знаний

База знаний содержит материалы по:

Процессам

Информационная безопасность

Технические вопросы

### Каталог услуг

Почта

Тестовые стенды

slack

инструменты

Тестирование

Промышленная среда