# Ведение версий

Платформа MyBPM состоит из двух разрабатываемых компонентов:

1. MyBPM WEB
2. MyBPM API

Каждый из этих компонентов имеет свой репозиторий в корпоративном хранилище. Ведение веток разработки у обоих этих компонентов одинаковое. Имеется следующий набор веток:

1. dev - главная ветка разработки.
2. dev-freeze - ветка фиксации текущего разработанного функционала - этап-альфа тестирования.
3. stage - стабильная тестовая версия - этап бета-тестирования.
4. release - ветка стабильных релизов.

## Процесс разработки

Вся разработка ведётся на ветке dev. Тут внедряются все новые функции, тестируются и исправляются ошибки. Мелкие функции вводятся напрямую, а функции по больше разрабатываются в отдельных ветках, ответвлённых от dev. После разработки и локального тестирования, ветки сливается в dev.

Например: разработчику по имени "Marat" получили сделать новую функцию, которую кратко можно назвать "zoo-to-settings". Он создаёт копию ветки dev и называет её:

dev-marat-zoo-to-settings

Далее он разрабатывает эту функцию в этой ветке. В процессе разработки он должен сливать изменения сделанные другими разработчиками из ветки dev в свою ветку dev-marat-zoo-to-settings.

После того как он разработал и протестировал новый функционал в этой ветке он сливает данную ветку в ветку dev. Если у него нет доступа на push в ветку dev, то он делает запрос на слияние merge-request. Обращается к старшему разработчику с целью проверки разработки, и, если всё нормально, старший разработчик применяет merge-request и новая функция попадает в ветку dev. Если у старшего разработчика есть замечания, то нужно эти замечания исправить и подойти снова за слиянием в общую ветку dev. После слияния в dev ветку dev-marat-zoo-to-settings необходимо удалить как локально, так и в репозитории.

## Процесс формирования релиза

После того как в ветку dev слили весь новый функционал текущего этапа разработки, ветка dev фиксируется тем, что она сливается в ветку dev-freeze. Процесс фиксации может затянуться из-за обилия ошибок, и в этот момент могут быть завершены новые функции, назначены на следующий этап, поэтому нужна новая ветка фиксации разработки. После того как функционал на ветке dev-freeze проверен и, в случае необходимости, исправлен, то ветка dev-freeze сливается в ветку stage. На этой ветке платформа начинает распространяться на тестовые среды клиентов. И, если на этом этапе выявляются ошибки, то они исправляются прямо в ветке stage - подобные действия называются hot-fix-ами.

Когда ветка stage окончательно стабилизирована и проверена на всех тестовых средах, то её сливают в ветку release.

## Процесс формирования версий

Версии системы состоят из четырёх чисел (на ветке dev-freeze - пять чисел), разделённых точкой, например:

4.11.3.117

Первое число (4) обозначает главную версию самой платформы. Главная версия платформы увеличивается не чаще одного раза в год.

Второе число (11) обозначает либо релиз системы, либо состояние тестирования. Если это число НЕЧЁТНОЕ, то данное число обозначает этап тестирования или разработки (stage, dev-freeze, dev). Если это число ЧЁТНОЕ, до это - РЕЛИЗ.

Третье число (3) обозначает уровень тестирования. Если оно НЕЧЁТНОЕ, то это альфа-тестирование или разработки (ветка dev или dev-freeze). Если ЧЁТНОЕ, то это бета-тестирование (ветка stage).

Четвёртое число (117) это номер текущего коммита. После каждого коммита, это число увеличивается на 1.

На ветке dev-freeze имеется ПЯТЬ уровней ведения версий. Первые четыре - это копия той версии с ветки dev которая замораживается. А последняя - это счёт фиксов в ветке для заморозки.

Первые числа нужно менять вручную. А последнее число увеличивается на 1 автоматически системой CI/CD, а уменьшается всегда до нуля в ручную.

Пример состояния версий на ветках:

1. dev - 4.11.7.118
2. dev-freeze - 4.11.5.121.3 - здесь видно какая прежняя версия в ветки dev была заморожена
3. stage - 4.11.6.3
4. release - 4.10.0.2

Во время разработки, изменения сбрасываются в ветку dev и последнее число автоматически увеличивается на единицу системой CI/CD.

### Исправления на ветке dev

Ветка dev является основной веткой разработки, поэтому любые изменения в ней не должны приводить к дальнейшим слияниям. После появления коммита на ветке dev автоматически запускается система CI/CD, и если нет ни каких ошибок в модульных тестах, и проходит нормально сборка, то последнее число версии (четвёртое) увеличивается на единицу.

### Исправления на ветке dev-freeze

Если исправляется ошибка на ветке dev-freeze, то она вносится на эту ветку коммитом, и пятое число увеличивается автоматически системой CI/CD на единицу.

После внесения коммита на ветку dev-freeze разработчик ОБЯЗАН слить изменения по схеме:

dev-freeze -> dev

**ВНИМАНИЕ**: При таком слиянии на ветках ВЕРСИИ МЕНЯТЬ НЕЛЬЗЯ - нужно оставлять такими, какие они были.

Например: если версии веток были следующие:

dev=4.11.7.118
dev-freeze=4.11.5.121.3

То сразу же после слияния dev-freeze -> dev они должны остаться такими же

dev=4.11.7.118
dev-freeze=4.11.5.121.3

После чего на ветке dev запуститься CI/CD и через некоторое время версия на dev увеличиться на единицу так:

dev=4.11.7.119
dev-freeze=4.11.5.121.3

### Хот-фиксы, или исправления на ветке stage

Если исправляется ошибка на ветке stage (это называется хот-фикс), то четвёрная цифра версии на ветке stage автоматически увеличивается на единицу. После этого разработчик ОБЯЗАН слить изменения по схеме:

stage -> dev-freeze
dev-freeze -> dev

При этих слияниях версии на ветках НУЖНО оставить неизменными.

После этих слияний запуститься CI/CD на обеих ветках, и последние числа версий на ветках dev и dev-freeze увеличиться на единицу.

### Супер-хот-фиксы, или исправления на ветке release

Если исправляется ошибка на ветке release (это называется супер-хот-фикс), то четвёрная цифра версии на ветке release автоматически увеличивается на единицу. После этого разработчик ОБЯЗАН слить изменения по схеме:

release -> stage
stage -> dev-freeze
dev-freeze -> dev

При этих слияниях версии на ветках НУЖНО оставить неизменными.

После этого запуститься CI/CD. По окончанию работы CI/CD последнее число версий на ветках dev, dev-freeze и stage увеличатся на единицу.

## Пример ведения версий

Изменения в ветке dev должны быть проверены и протестированы. При этом не должен остановиться процесс разработки новых функций на платформе. Для этого предусмотрен следующий механизм ведения версий:

### Заморозка ветки dev

Когда в ветке dev весь необходимый на данном этапе функционал реализован, то нужно его заморозить. Для этого необходимо все изменения в ветке dev слить в ветку dev-freeze вместе с самой версией.

Например, имеется состояние версий:

1. dev - 4.11.7.118
2. dev-freeze - 4.11.5.34.40
3. stage - 4.11.6.3
4. release - 4.10.0.2

Поступила команда от руководителя проекта на заморозку ветки dev, то изменения сливаем из dev в dev-freeze и в ручную устанавливаем следующие версии:

1. dev - 4.11.7.118
2. **dev-freeze** - **4.11.7.118.0**
3. stage - 4.11.6.3
4. release - 4.10.0.2

Т.е. на ветку dev-freeze ПРОСТО копируем версию с ветки dev и добавляем ПЯТОЕ число равное нулю.

Дальне система CI/CD отрабатывает и увеличивает пятое число до единицы, как показано ниже:

1. dev - 4.11.7.118
2. dev-freeze - 4.11.7.118.**1**
3. stage - 4.11.6.3
4. release - 4.10.0.2

Дальше на dev-freeze-е могут быть найдены ошибки и исправлены прямо на этой ветке. Их может быть несколько. После нескольких исправлений может возникнуть следующая картина:

1. dev - 4.11.7.**231**
2. dev-freeze - 4.11.7.118.**18**
3. stage - 4.11.6.3
4. release - 4.10.0.2

Т.е. было как минимум 18 коммитов в ветку dev-freeze о чём свидетельствует пятая цифра в версии.

Так же заметьте, что у ветки dev, тоже увеличилась четвёртая цифра, так как в неё влили все изменения из ветки dev-freeze, да и ещё в ней могли появиться новые функции, предназначенные для следующего релиза.

### Стабилизация нового функционала: обновление ветки stage

После того как ветка dev-freeze стабилизирована, поступает команда от руководителя проекта залить её в ветку stage. Исполнив эту команду, должны быть следующие изменения в версиях:

Было:

1. dev - 4.11.7.231
2. dev-freeze - 4.11.7.118.18
3. stage - 4.11.6.3
4. release - 4.10.0.2

Стало:

1. dev - **4.11.9.0**
2. dev-freeze - 4.11.7.118.18
3. **stage** - **4.11.8.0**
4. release - 4.10.0.2

Здесь показано, что в ветке stage увеличилась третье число на два (с 6 до 8 - должны быть ЧЁТНЫМИ), а четвёртое занулили.

Так же изменили версию в ветке dev - тоже увеличили третье число на два (с 7 до 9 - должны быть НЕЧЁТНЫМИ)

После этих коммитов запуститься CI/CD на ветках dev и stage и последние версии увеличатся на единицу:

1. dev - 4.11.9.**1**
2. dev-freeze - 4.11.7.118.18
3. stage - 4.11.8.**1**
4. release - 4.10.0.2

А после нескольких исправлений может быть следующая картина:

1. dev - 4.11.9.**17**
2. dev-freeze - 4.11.7.118.18
3. stage - 4.11.8.**6**
4. release - 4.10.0.2

Ветка dev-freeze уже не трогается и она не меняется, а хот-фиксы изменяют последнее число в версии ветки stage. При последующем сливании этих хот-фиксов в ветку dev последнее число на этой ветке тоже увеличивается на единицу. Если кто-то сольёт изменения в ветку dev-freeze, то на ней тоже последнее число увеличиться на единицу - это делать не обязательно, но не возбраняется.

### Выпуск релиза: обновление ветки release

После того как ветка stage стабилизирована и хорошенько оттестирована, то от руководителя проекта поступает команда на переход изменений в ветке stage в ветку release. Должно быть сделано следующее:

Было:

1. dev - 4.11.9.2071
2. dev-freeze - 4.11.7.118.31
3. stage - 4.11.8.11
4. release - 4.10.0.2

Стало:

1. dev - **4.13.1.0**
2. dev-freeze - 4.11.7.118.31
3. stage - **4.13.2.0**
4. release - **4.12.0.0**

Т.е. вышел новый релиз - **4.12.0.0**!!!

Тут увеличилось второе число на ветке release на два (с 10 до 12 - должно быть ЧЁТНЫМ) - остальные были сброшены до 0.

На ветках dev и stage второе число было установлено на единицу БОЛЬШЕ релиза (13 - НЕЧЁТНОЕ).

Третье число на ветке dev стало единицей (нечётным), а на ветке stage стало двойкой (чётное).

Четвёрное число на ветках dev и stage обнулили.

На ветке dev-freeze ни каких изменений НЕТ - там лишь указывается какая ветка была когда-то заморожено. Так будет до тех пор, пока опять не заморозят ветку dev.

После этих изменений на всех ветках запуститься CI/CD, и когда он везде отработает, то будут такие изменения в версиях:

1. dev - 4.13.1.**1**
2. dev-freeze - 4.11.7.118.31
3. stage - 4.13.2.**1**
4. release - 4.12.0.**1**

## Инварианты

Инварианты - это условия, которые не должны соблюдаться при каждой операции push.

### Терминология ведения версий

Версия на ветке представляет собой набор чисел разделённых точкой. Первое число - это первый уровень, второе - второй, и так далее.

### Базовые инварианты

1. Чётное число - это более стабильная версия чем нечётное число на своём уровне ведения версий. Исключение первый уровень и последний.
2. Нестабильные ветки всегда больше по версии чем стабильные - так как они впереди разработки.

### Основные инварианты

1. На ветках dev, stage, release - четыре уровня версии.
2. На ветке dev-freeze - пять уровней версии.
3. На ветке release второй уровень всегда чётный.
4. На остальных ветках второй уровень всегда нечётный.
5. На ветке release второй уровень всегда на 1 меньше, чем на остальных (исключение dev-freeze).
6. На ветке release третий уровень всегда равен нулю.
7. На ветке stage всегда третий уровень чётный.
8. На ветке dev всегда третий уровень нечётный.
9. На ветке dev всегда третий уровень на 1 больше, чем на ветке stage.
10. На ветке dev-freeze всегда пять уровней. Первые четыре - это точная копия некого старого dev-а.

# Описание API-сервисов

Описание API-сервисов можно найти по пути:

[API-сервисы CRUD](https://gitlab.com/ekolpakov/mybpm-doc/-/blob/master-on-inet/doc/api-service/101-crud.md)

[Видео для работы с API](https://www.youtube.com/watch?v=BUEHXiyIQ-8)

[Путь к апи](090_api-service)

# Настройка клиентского плагина MyBPM

1. Подключаем нужные библиотеки из репозиториев **"jsrepo.greetgo"**
2. Подготавливаем нужные компоненты плагина
3. Добавляем компоненты, модули в **webpack.config.js** и **docker/lib/plugins-config.json**
	1. Структура **"plugins-config.json"**
	2. **"pluginProjectName"** название проекта плагина, uniqueName из webpack.config.js
	3. **"path"** путь до plugin.js
	4. **"plugins"** массив плагинов находящихся в этом проекте
	5. **"plugins.key"** уникальный айди плагина
	6. **"plugins.type"** тип плагина
	7. **"plugins.exposedModule"** название модуля которое ранее указали в **webpack.config.js**
	8. **"plugins.ngModuleName"** имя компонента или модуля в плагине
	9. **"plugins.route"** используется при type === 'PAGE', роутинг плагина
4. Общие зависимости нужно прописать в **webpack.config.js**
5. Запускаем скрипт **"plugin-config-copy-to-debug.bash"** после чего запускаем **mybpm-web**
6. Проверяем подключился ли плагин
7. Коммит, пуш

Доступные типы плагинов, на данный момент:

**CONSOLE** - Текст в консоль

**MENU\_ITEM** - Добавление меню в список

**PAGE** - Отдельная страница

**OVERLAY** - Оверлей на главный экран

**IN\_OUT** - Взаимодействие между ядром и плагином

**COMPANY\_PAGE\_HEADER\_CONTENT** - Контент в шапку страницы компании

**BO\_VIEWER\_HEADER\_CONTENT** - Контент в шапку реестра бо

**REGISTER\_PAGE\_BOTTOM\_CONTENT** - Контент под кнопку в регистрации

**AUTH\_ADDITIONAL\_LOGO** - Внешний логотип на странице авторизации

**HEADER\_ADDITIONAL\_LOGO** - Внешний логотип в шапке системы

# Инструкция по скачиванию логов с клиента

При возникновении ошибки нужно выгрузить логи с клиента и передать их на исправления.

Для этого нужно сделать следующее:

1. Открыть dev-tools во вкладке MyBPM (F12)
2. Перейти во вкладку console в dev-tools
3. Скопировать и вставить команду в консоль - window['AMS\_UNLOAD']()
4. Дождаться скачивания текстового документа
5. Передать скаченный файл на анализ разработчикам

Дополнительно: При возникновении ошибки желательно не перезагружать страницу MyBPM, для того чтобы логи сохранили более точную последовательность действий, и разработчикам было легче понять что произошло.

# Рекомендации при работе с elasticsearch

## Распределение памяти для elasticsearch

* При распределении памяти, необходимо выделить память для самого контейнера elasticsearch *memlimit*, в зависимости от количества выделенной памяти для контейнера, нужно добавить переменную окружения ES\_JAVA\_OPTS, память, выделенная через данную переменную, будет использована для системного кэша.
* **Важно:** память, которая выдается через ES\_JAVA\_OPTS, не должна превышать 50% доступной ОЗУ, выделенной на контейнер, потому что вся оставшаяся память будет выделена для [Apache Lucene](https://opster.com/guides/elasticsearch/glossary/elasticsearch-lucene/).
* Подробнее можно прочитать [здесь](https://opster.com/guides/elasticsearch/capacity-planning/elasticsearch-memory-usage/).
* В зависимости от потребностей, можно выделить менее 50% памяти, вся оставшаяся память будет выделена для Apache Lucene, что может улучшить производительность. Данная практика советуется, если не производятся операции, связанные с агрегированием строк *(fielddata)*.
* Подробнее можно прочитать [здесь](https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html#_give_less_than_half_your_memory_to_lucene).
* Для лучшей работы *heap size* для elasticsearch не должен превышать [30GB](https://www.elastic.co/guide/en/elasticsearch/reference/8.8/advanced-configuration.html#set-jvm-heap-size), так как при использовании меньшей памяти, используются [сжатые указатели.](https://www.elastic.co/guide/en/elasticsearch/reference/7.11/advanced-configuration.html#setting-jvm-heap-size)
* Дополнительно можно ознакомиться [здесь](https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html#compressed_oops).
* В идеале использовать не больше 4GB heap sizing, так как будут использованы [32-битные указатели](https://www.elastic.co/blog/a-heap-of-trouble).
* Выделяемая память для elasticsearch должна быть как можно меньше, не рекомендуется выделять много памяти, так как это может привести к более долгим паузам [сборщика мусора](https://www.elastic.co/blog/a-heap-of-trouble).
* При выделении малого количества памяти сборщик мусора не будет освобождать ресурсы в достаточном количестве, что приведет к [переполнению памяти](https://www.elastic.co/blog/a-heap-of-trouble). Также при выполнении запросов, если сервер ноды кластета elasticsearch достигает состояния, при котором происходит нехватка ресурсов, сервер может перестать отвечать, [во избежание полного отключения](https://opster.com/guides/elasticsearch/operations/elasticsearch-circuit-breakers/).

## Мониторинг сборщика мусора

* Нормальное состояние работы сборщика мусора заключается в частых и периодических срабатываниях освобождения ресурсов в достаточном количестве. Используемая память при работе elasticsearch варьируется от 30% до 70% от общей доступной памяти, выделенной на JAVA процесс ноды, находящейся в кластере elasticsearch, [при нормальном состоянии](https://opster.com/guides/elasticsearch/capacity-planning/elasticsearch-heap-size-usage/).
* Нагрузка на память не должна превышать в стабильном состоянии 75%. При высокой потребности нагрузка не должна превышать [85%](https://www.elastic.co/blog/found-understanding-memory-pressure-indicator/#green-is-good).
* При нагрузке выше чем [85%](https://www.elastic.co/blog/found-understanding-memory-pressure-indicator/#is-high-memory-pressure-always-a-problem) стоит провести мониторинг работы сервера elasticsearch, так как это может привести к дальнейшим [проблемам](https://www.elastic.co/guide/en/cloud/current/ec-metrics-memory-pressure.html).
* Возможные решения можно просмотреть [здесь](https://www.elastic.co/guide/en/cloud/current/ec-memory-pressure.html).
* **Заметка:** чтобы проверить нагрузку на ноды, можно сделать следующий запрос
* GET {host}/\_cat/nodes?v&h=name,node\*,heap\*
* где *host* - это хост ноды elasticsearch.

## Настройка репликации

* При работе состояние кластера должно иметь статус *Green*, это означает, что для каждого основного шарда, количество созданных репликационных шардов соответствует настройке количества репликационных шардов для всех индексов. В ином случае статус будет [желтый или красный](https://www.elastic.co/guide/en/elasticsearch/reference/current/red-yellow-cluster-status.html).
* Количество создаваемых реплик не должно быть больше или равно количеству нод в кластере elasticsearch, так как репликационный шард не может создаваться на той же ноде, на которой присутствует основной шард. Подробнее можно прочитать [здесь](https://www.elastic.co/guide/en/elasticsearch/reference/current/diagnose-unassigned-shards.html) или [здесь](https://www.elastic.co/blog/managing-and-troubleshooting-elasticsearch-memory).
* Настройку количества репликационных шардов можно проверить в настройках индекса по пути:
* index.number\_of\_replicas

## Шардирование (Sharding)

* Количество шардов на каждой ноде может быть неограниченным, но для лучшей производительности, на 1GB памяти JVM, иметь не более [20 шардов](https://www.elastic.co/blog/managing-and-troubleshooting-elasticsearch-memory).
* Также рекомендуется иметь шарды размером от [20GB до 40GB](https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html#it-depends). Иметь шарды меньшего размера не [оптимально](https://www.elastic.co/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster).
* Также стоит настроить [ILM](https://www.elastic.co/guide/en/elasticsearch/reference/current/index-lifecycle-management.html) для решения проблемы с большим количеством [шардов](https://opster.com/guides/elasticsearch/capacity-planning/elasticsearch-oversharding/).