[Архив] Процедура сборки и развертывания

История изменений

Дата	Версия	Описание	Автор
10.04.2020	1.0	Черновик по результатам совещания 08.04	Karinski, Andrey

Черновик!

Введение

Документ описывает общую процедуру передачи исходного кода для сборки и деплоя продуктов в контуре ГПН.

Цель

Цель документа - донесение до всех заинтересованных лиц (прежде всего разработчиков и инженеров по развертыванию) общей концепции передачи исходного кода заказчику.

Область применения

Распространяется на всех специалистов, которые работают с исходным кодом продуктов (тимлид проекта, сервис инженер, системным администраторам и т.п., в том числе на стороне ГПН).

Определения и сокращения

Все использованные в тексте акронимы хорошо знакомы читателю.

Ссылки

SemVer

Обзор

Описывает, как организован данный документ.

Процедура сборки

Общие положения

В связи с требованиями безопасности исходный код продуктов должен передаваться на сторону ГПН, где будет проверяться на уязвимости, собираться и развертываться.

Текущая версия документа не описывает процедуру развертывания!

Кратко рабочий цикл от передачи кода до размещения в хранилище артефактов можно описать так:

1. В ГПН создается форк репозитория GXT.
2. Код из ветки release/xxx или (в некоторых случаях) из ветки master вытягивается в соответствующую ветку репозитория ГПН.
3. На сторону ГПН передается список зависимостей проекта.

Подготовка к сборке

Синхронизация репозиториев

Как уже указывалось ранее, обязательное требование ГПН - сборка должна выполняться на ее стороне, следовательно, требуется обеспечить дублирование репозиториев и периодическую их синхронизацию. Почти всегда код будет передаваться от GXT в ГПН, но в отдельных случаях (например, исправление конфигурационных файлов, добавление тестовых датасетов и т.п.) необходимо передавать изменения из ГПН в GXT. Очевидно, что в последнем случае такие изменения обязаны пройти процедуру `code review`, а значит и передаваться должны посредством `pull request`.

Рассматривается два сценария взаимодействия репозиториев на стороне GXT и ГПН:

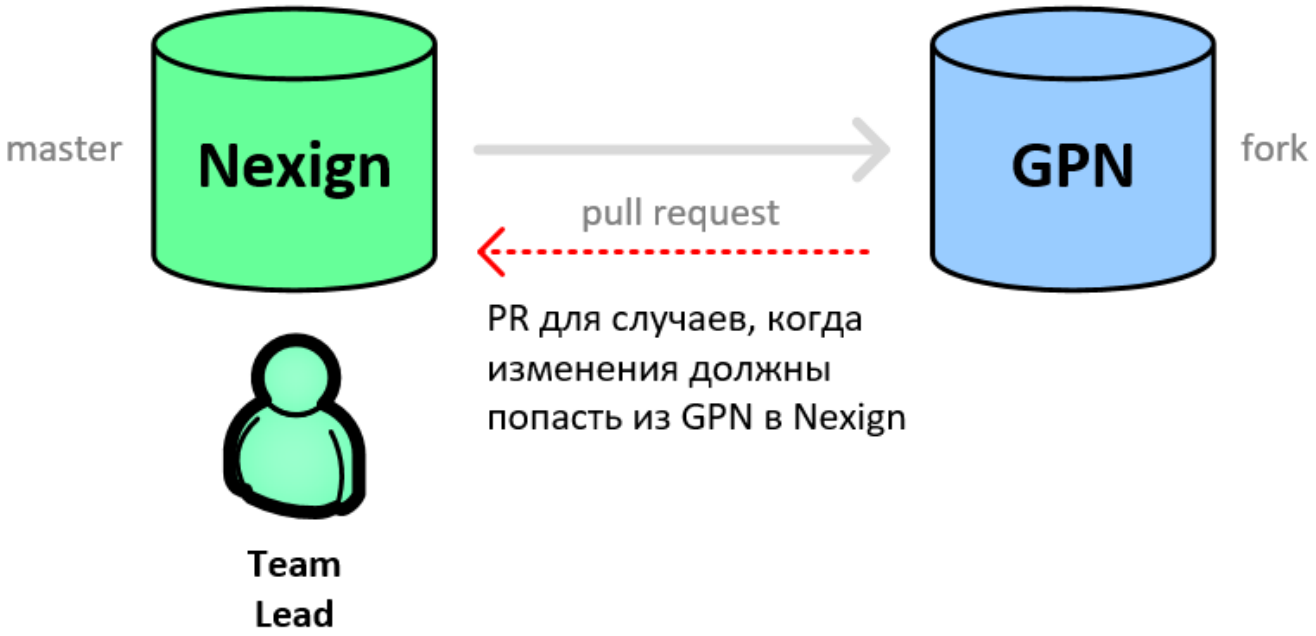
- Мастер GXT и форк GPN
- Мастер GPN и форк GXT

Оба подхода имеют право на жизнь, однако рабочим вариантом видится первый.

Мастер GXT и форк GPN

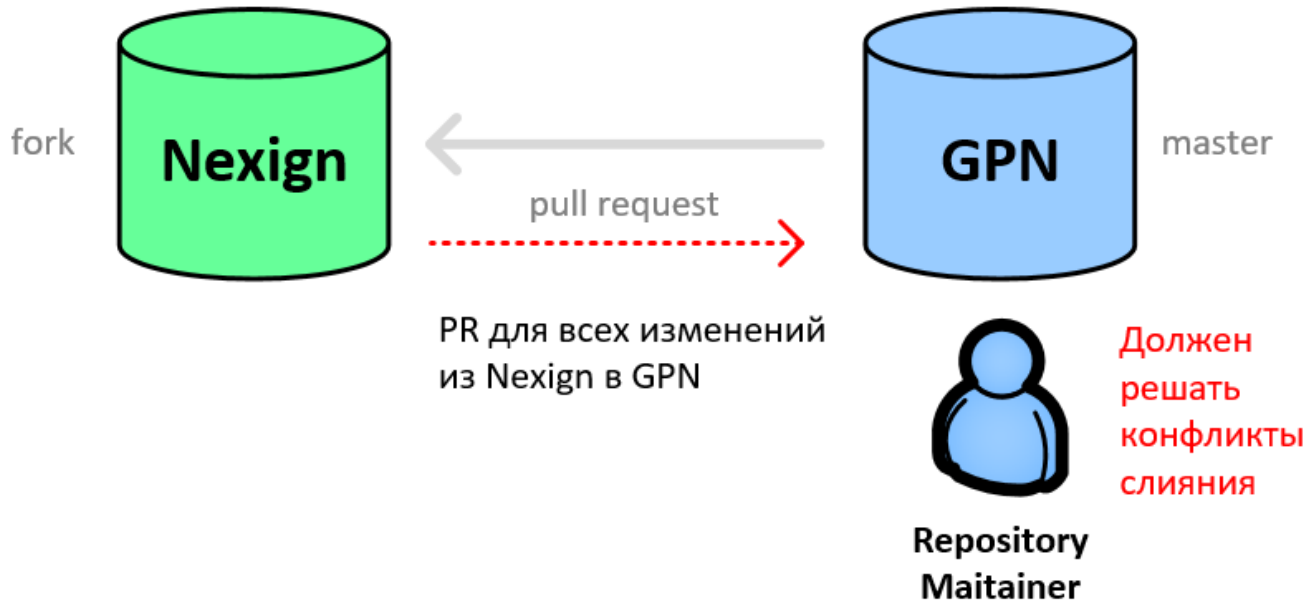
Очевидно, в случае, когда владельцем репозитория является команда-разработчик, процедура синхронизации репозиториев логична:

1. Код из мастер-репозитория попадает в форк автоматически, по событию коммита или расписанию.
2. Изменения из форка в мастер попадают только через `pull request`, при этом `team lead` проводит `code review` всех изменений.



Мастер GPN и форк GXT

Альтернативный сценарий синхронизации предложен из-за того, что ГПН предполагает в некоторых проектах использовать несколько независимых команд, следовательно, будет логичным разместить мастер-репозиторий в ГПН. Однако здесь появляется существенная проблема - необходимость разрешать конфликты в коде, что требует исключительного понимания технических деталей проекта.



Модель ветвления GitFlow

За основу взята модель ветвления, которую Vincent Driessen описал в своей статье "A successful Git branching model" (перевод на русский). Нужно ли вносить изменения в git flow для разработки. Например, что бы иметь возможность выпуска минорых версих версий предыдущих мажорных версий продукта (Пример: image2020-4-15_14-59-36.png)? Если у нас forward only, то в этом нет необходимости.

- Версионность
  - Формат SemVer
  - Версонирование продукта
  - Управление версиями и Git Flow
  - Виды веток
  - Ветви master и develop
  - Ветвь feature
  - Ветвь hotfix
  - Ветвь release
  - Номера версий и ветки GitFlow

Автоматизация версионирования

Версионность

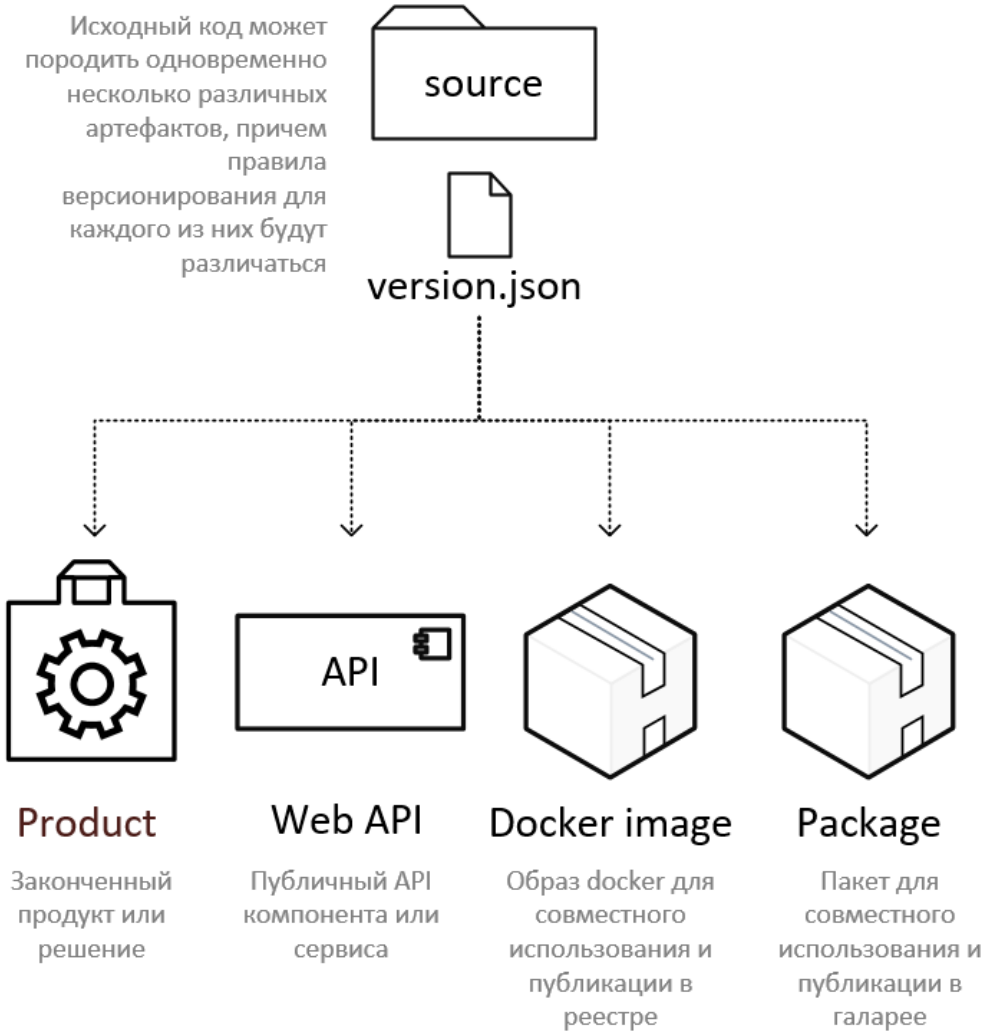
При разработке даже относительно несложных продуктов нужно быть уверенным в том, кто релиз никогда не будет выпущен прежде, чем пройдет все стадии анализа и тестирования.
Инструментов, нужные для выполнения этих задач, давно существуют, но для автоматизации их применения требуются некоторые соглашения и шаблоны.

Как правило, в жизненном цикле продукта важную роль играет его версия: она помогает понять, что, когда, кем и в какой последовательности менялось. Например, необходимо убедиться, что конкретная версия продукта может быть развернута только в тех средах, где она способна работать. Для этого потребуется способ уникально идентифицировать версии продукта (и, разумеется, всех артефактов проекта).

Нужно отметить, что проблема версионирования довольно объемна и распространяется одновременно на разные части продукта (и проекта, порождающего продукт), причем разные части продукта (и разные артефакты проекта соответственно) имеют свои истории версий со своими правилами и особенностями. Однако в данной статье будет рассматриваться только версионирование продукта.

В качестве решения данной проблемы рекомендуется простой набор правил и требований, определяющих, как назначаются и увеличиваются номера версий.

В момент создания проекта ему присваивается его первая версия в формате, рассматриваемом ниже. Любые изменения в проекте (добавление-удаление файлов, изменение их атрибутов, исправление исходного кода) отражаются особым изменением номера версий или метаданных в версии продукта.



version.json

```
{
  //     semVer.
  //          .
  "version":"3.0.0-beta.1+meta",
    //       ,
    "vcs": {
          "sha":"c4eb984a439f2b17c302166ed6b23048c207bffa",
      "branch": "master"
            "commitDate":"2020-04-20T11:10:34.164Z",
    },
  // ,
   "build": {
      "url": "https://tc/viewLog.html?buildId=6965196"
    }
}
```

Формат SemVer

Семантическое версионирование (Semantic Versioning) - схема версионирования, которая однозначно описывает смысл внесенных в продукт изменений и их историю.

На иллюстрации показаны семантические части строки версии. Наиболее важная из них первая часть **Major.Minor.Patch**, а полностью семантическая версия может состоять из следующих частей:

- **Major** - старший номер версии. Увеличивается, когда API библиотеки или приложения меняется обратно несовместимым образом.
- **Minor** - младший номер версии. Увеличивается, когда в API добавляются новые функции без нарушения обратной совместимости. При увеличении номера **Major** нумерация **Minor** сбрасывается в 0, то есть за версией 1.5 последует версия 2.0… а не 2.5
- **Patch** - номер патча (или номер сборки). Увеличивается при исправлении ошибок, рефакторинге и прочих изменениях, которые не меняют функционал. При увеличении **Major** или **Minor** в 0 сбрасывается **Patch**. В сценариях автоматической сборки **Patch** воспринимается как номер сборки (**Build**) и увеличивается при каждой сборке не 1, что в итоге позволяет легко отличать билды во время тестирования.
- **Pre-release** - необязательный, разделенный точками список, отделенный от трех номеров версии знаком **минус** (например: "1.2.3-beta.2"). Используется вместо тегов, чтобы помечать определенные вехи в разработке. Обычно это *alpha*, *beta*, *release candidate* и производные от них.
- **Metadata** - метаданные системы сборки. Разделен на составные части точками, но отделен от номеров версии или тегов плюсом (**+**). Данную информацию с точки продуктовой версии принято игнорировать, она интересна лишь разработчикам. В метаданные удобно включать имя ветки, из которой была собрана версия и хэш-код коммита. Например, 0.1.0-alpha.1+Branch.develop.Sha.cdb698d7521ef32.

Еще раз, простое правило версионирования:

Учитывая номер версии **Major.Minor.Patch**, следует увеличивать:

- **Major** - когда сделаны обратно несовместимые изменения API.
- **Minor** - когда добавлена новая функциональность, не нарушая обратной совместимости.
- **Patch** **-** когда делаются обратно совместимые исправления.
- **Pre-release** (опционально) - тэг в зависимости от фазы жизненного цикла продукта (alpha, beta), а номер билда - при каждом изменении.
- **Metadata** (опционально) - автоматически в процессе сборки.

Еще несколько правил:

- Начальная версия каждого нового продукта - 0.1.0
- В момент первого релиза ему присваивается версия 1.0.0
- Перед удалением из проекте части функционала необходимо выпустить промежуточный минорный релиз (например, 1.0.0 → 1.1.0), в котором функционал, запланированный к удалению в версии 2.0.0, будет помечен как устаревший.
- В строке версии может использоваться префикс "**v**" (например, v1.0.0), но его принято игнорировать.

Более подробно о правилах версионирования можно узнать по ссылке semver.org

Версонирование продукта

Для версионирования продукта используется следующее правило.

Мажораня версия:

Prototype → 1

MVP → 2

OFP → 3

OFP+ и т.д. → необходимо руководствоваться Semver, здравым смыслом и потребностями продукта.

Минорная версия и патч соотвествуют правилам SemVer, но обычно используется правило:

Минорная версия:

Релиз спринта → +1

Патч:

После релиза версии и установки на прод для текущей версии увеличивается только патч.

Управление версиями и Git Flow

Рабочий процесс Git Flow представляет собой модель ветвления с жесткими границами между ветвями и особыми правилами переноса кода. Основные принципы довольно просты и интуитивно понятны.
Одним из преимуществ этой модели является то, что она очень хорошо работает с SemVer и моделью непрерывной интеграции и развертывания (CI/CD). От читателя ожидается, что он уже знаком с Git Flow, и документ описывает совместное использование Git Flow и SemVer.

Виды веток

Сборки для тестирования рекомендуется брать из веток **develop** или **release/${version}**.

№	Ветка	Описание
1	**master**	Описание: основная ветка хранения продукта, содержащая последнюю по нумерации отгруженную версию; необходима для хранения линейной истории развития продукта, создания новых версий продукта с чистой историей коммитов путем перемещения в нее требуемой версии. Содержит **merge** из веток: **release/${version}**, **develop**.
2	**develop**	Описание: ветка предназначена для разработки определённой версии/версий продукта. Вместо имени **develop** может быть использована версия продукта в формате SemVer, если есть необходимость поддерживать несколько мажорных версий. Примеры названий веток: • **develop** • **1.x.x** • **1.5.x** • **1.5.3** Ответвляется от веток: **master**, **1.x.x**, **release/${version}**. Содержит **merge** из веток: **feature/*, bugfix/*, hotfix/***. Содержит прямые **commit'ы**: нет
3	**release/${version}**	Описание: ветка предназначена для подготовки к релизу версии/версий продукта. После релиза данная ветка замораживается для изменений. Примеры названий веток: • **release/3.2.1** Ответвляется от веток: **develop**. Содержит прямые **commit'ы**: нет.

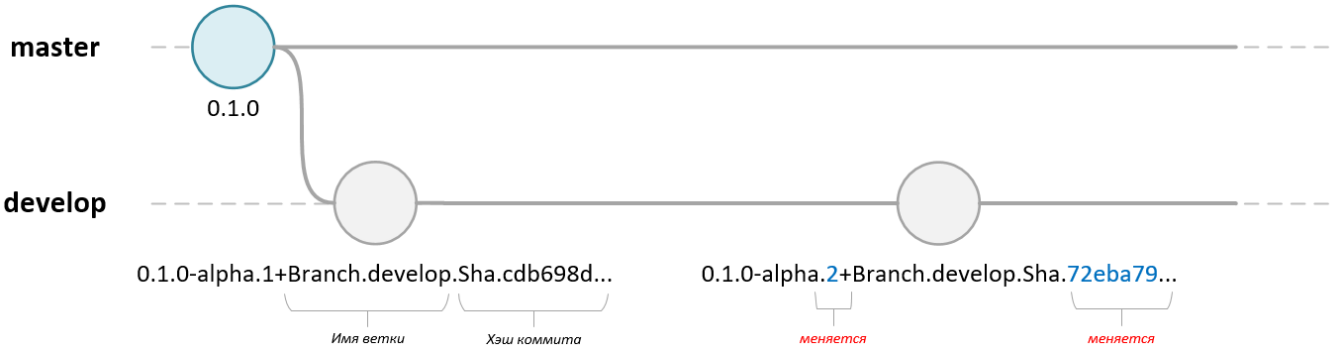
4	**feature /*** **bugfix/*** **hotfix/*** **refactor /***	Описание: ветка разработки фрагмента функциональности, исправления ошибки и проведения рефакторинга в версии продукта. Правило именования ветки: **feature/${version}/${jira-ticket-key}/${short-description}**, где: • **${version}** – например, когда функциональность по одному и тому же тикету реализуется в двух разных версиях в двух разных ветках по-разному; • **${jira-ticket-key}** – не указывается **только** в случае отсутствия тикета на работу или **${jira-ticket-key}** указан в **commit**'ах, что позволяет организовать корректную связь с **jira**; • **${short-description}** – краткое описание изменений Примеры названий веток: • **feature/3.2.1/PROJECT-4511/add-new-code** • **bugfix/PROJECT-1766/my-module/memory-leak** • **refactor/rename-a-to-b** • **hotfix/2.2.0/PROJECT-123/PROJECT-12345/crash-app**

Ветви master и develop

Эти две ветви долговечны и должны существовать всегда. Каждый из них представляет историю проекта в разных состояниях.
Ветка **master** представляет официальную историю релизов. Каждый новый релиз будет объединен с **master**, в этой ветке находится последнее официально опубликованное состояние проекта.
Ветка **develop** служит для интеграции функционала и содержит полную историю проекта. Новые ветки релиза должны быть созданы из ветки разработки и будут использоваться как временная ветка для подготовки новой версии.
Ветка **master** должна регулярно сливаться с **develop**.

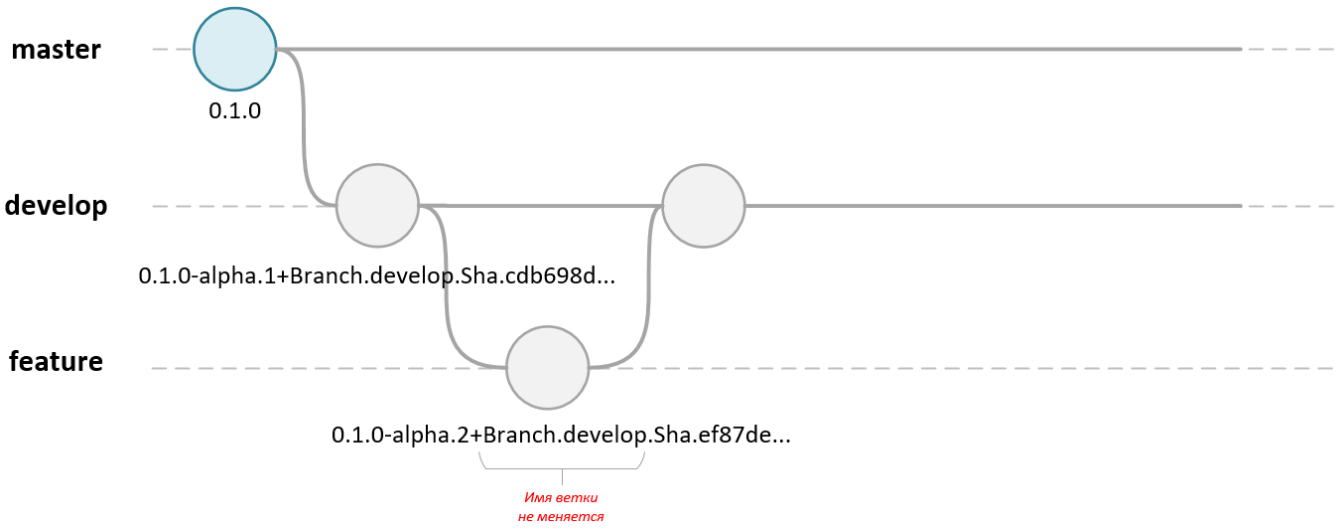
В примере ниже и в соответствии с правилами SemVer, версия **develop** получается из ветви **release** с добавлением к ней префикса **pre-release** и номера билда.

Удобно включить в версию имя ветки, на основе которой она создалась, и хэш коммита. Лучше автоматизировать этот процесс через конвейер CI или сборочным скриптом на стороне разработчика. На рисунке ниже можно заметить, как между версиями меняется номер билда и хэш (здесь и далее изменения отмечены синим цветом).



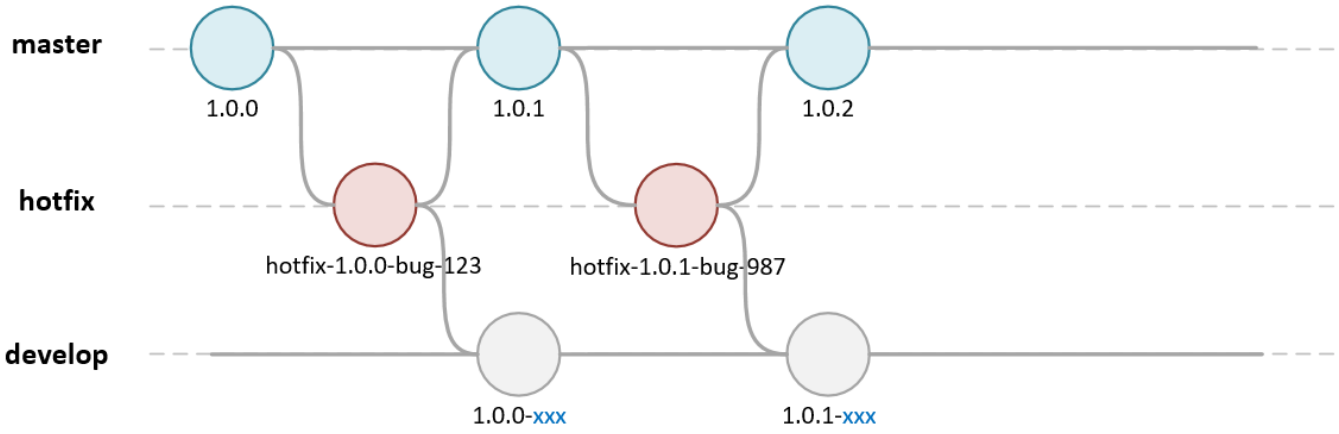
Ветвь feature

Каждая функция должна находиться в отдельной ветке, пока она не будет завершена, но так же стоит декомпозировать код таким образом, что бы функции были небольшими, а количество изменяемых файлов минимальным. Когда функция завершена, она объединяется с **develop** и готовой стать частью следующей релизной ветки. Номер версии **feature**-ветки может всегда имеет **pre-release**-префикс *alpha*, а версия ветки **devel op** может иметь префикс *alpha* или *beta* (предпочтительно, но не обязательно). Политику выдачи префиксов можно автоматизировать.



Ветвь hotfix

Ветка **hotfix** используется для исправлений к релизам. Эти исправления должны быть сделаны с осторожностью, и ответственность за их негативное влияние на производственную среду несет разработчик и группа тестирования.
Это позволяет команде разработчиков выполнять итерации выпусков, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла выпуска. Исправление ответвляется от определенного релиза из **matser**, а после завершения работы сливается с **master** и **develop**.
Номер версии исправления формируется из версии релиза и включает идентификатор задачи в баг-трекере (например, Jira).
Только сейчас заметил, что я сначала сливал hotfix с master, а потом master с develop

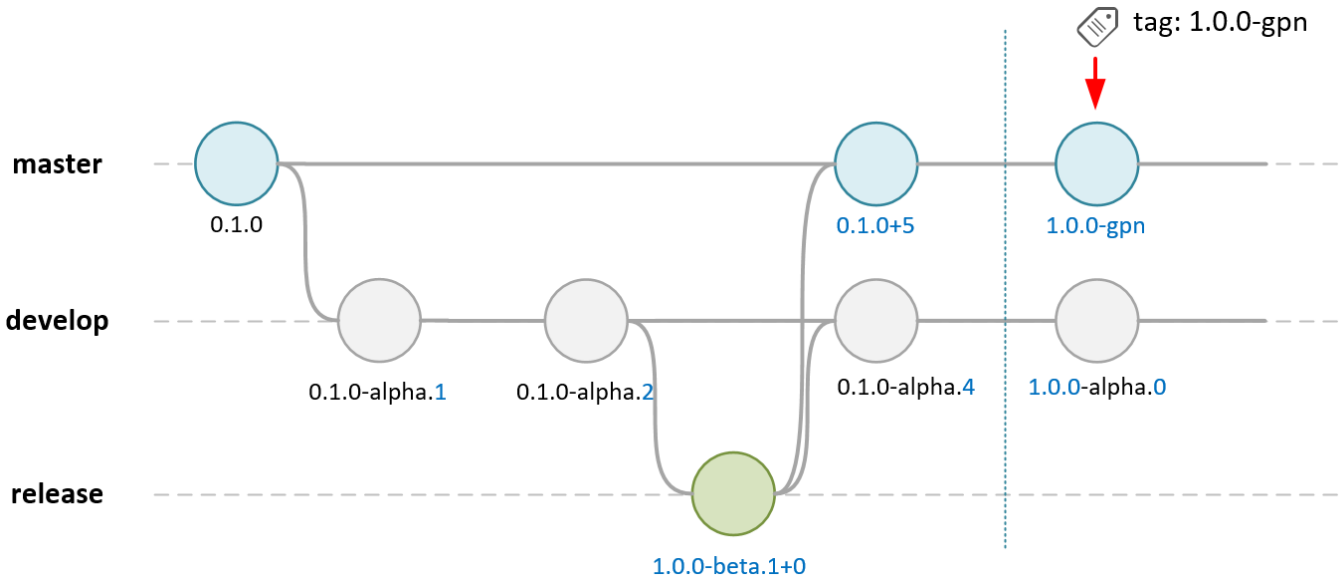


Ветвь release

В определенный момент (при приближении заранее определенной даты выпуска) или при других обстоятельствах, создается новая ветка **release /xxx** для подготовки и стабилизации новой версии продукта. После создания ветки релиза новые функции вводить не следует, и допускаются только изменения в форме исправлений ошибок, незначительных улучшений, правки документации и других задач, ориентированных на решение проблем выпуска.
Команда разработчиков работает в тесном контакте с командой поддержки, чтобы перебрать проблемы в ветке релиза и обработать ее, пока она не будет считаться готовой к работе.

1. При создании ветки release на основе develop ее версия имеет pre-release-префикс beta.

2. После внесения необходимых измененияx в релизную ветку и слияния ее с master к последней применяется тэг с финальной версией 1.0.0-gpn.
3. Теперь версия master будет иметь версию 1.0.0-gpn



Номера версий и ветки GitFlow

Примеры выше показали наличие некоторых правил, на основе которых версиям присваиваются **pre-release**-префиксы. В основном правило выглядит так: версия master не содержит префиксов (потому что она финальная), версия release и hotfix имеет префикс release candidate,

Правила присвоения версий в зависимости от ветки:

Ветка	Тип релиза	Формат версии	Пример ветки	Тэг	Пример версии
feature	*alpha*	**Major.Minor.Patch-**alpha**.Build.Feature**	feature/super-overkill-feature		1.1.0-alpha.1
develop	*beta, [alpha]*	**Major.Minor.Patch-**alpha|beta**.Build**	develop		1.1.0-beta.1
release	*release candidate, [beta]*	**Major.Minor.Patch-**beta|rc1	release/1.1.0		1.1.0-rc.1
hotfix	*release candidate*	**Major.Minor.Patch-**rc1**.Build**	hotfix/1.1.0/bug-123		1.1.0-rc.2
master	*stable*	**Major.Minor.Patch**	master	v1.1.0	1.1.0

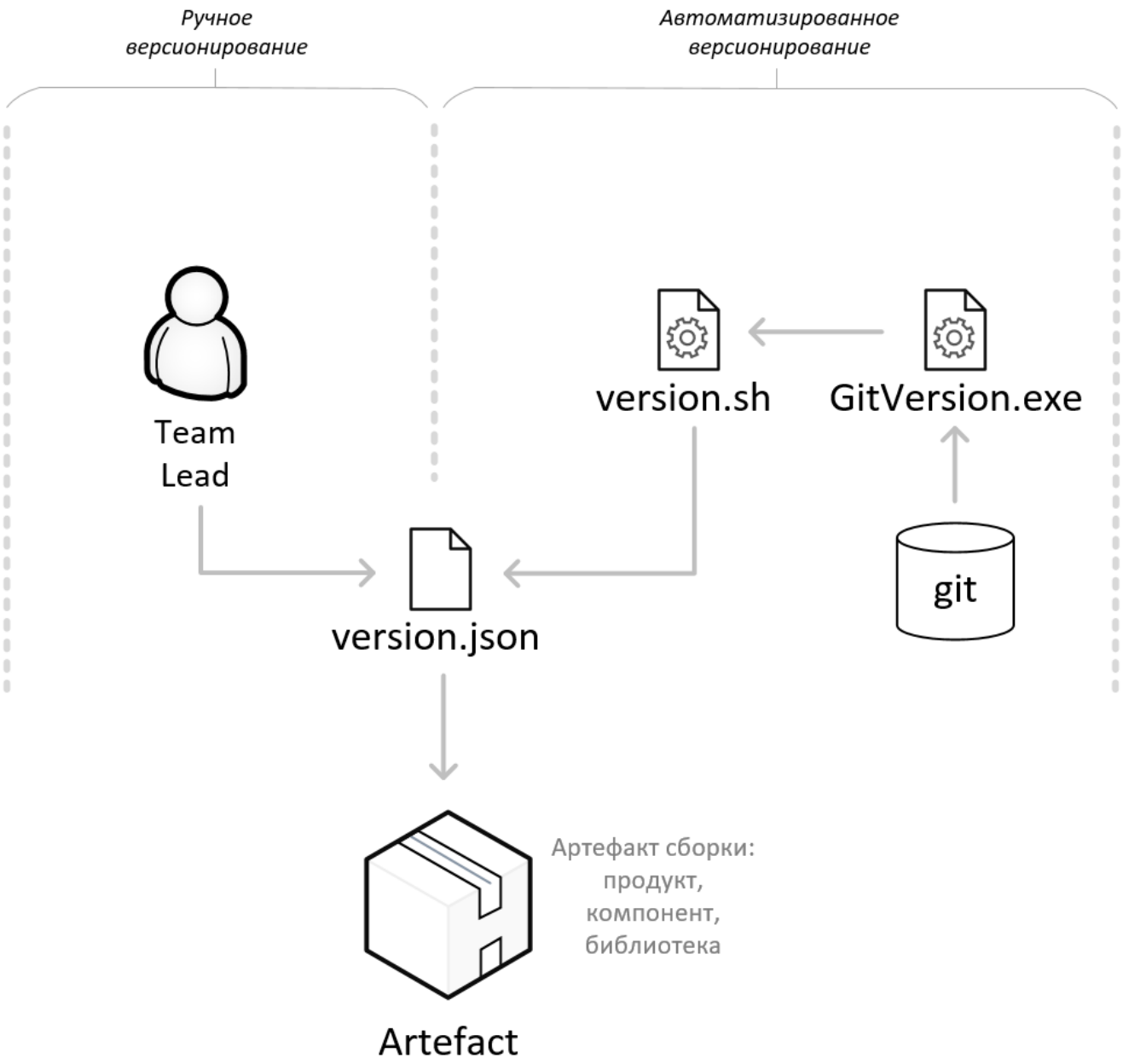
На рисунке ниже показано, как

Автоматизация версионирования

Для версионирования (то есть задания значения текущей версии в файле version.json) можно использовать два равнозначных подхода: ручной и автоматизированный.

- При ручном - разработчик (вероятнее всего тимлид) вручную задает значение поля SemVer согласно правилам, описанным о предыдущих частях документа.
- При автоматизированном версионировании заполнение файла version.json выполняется скриптом version.sh, который в свою очередь берет номер версии (и ее метаданные) из утилиты наподобие GitVersion, а утилита извлекает эти данные из локального репозитория git.

Но даже если выбран ручной режим, разработчику ничто не помешает запустить скрипт version.sh самостоятельно.



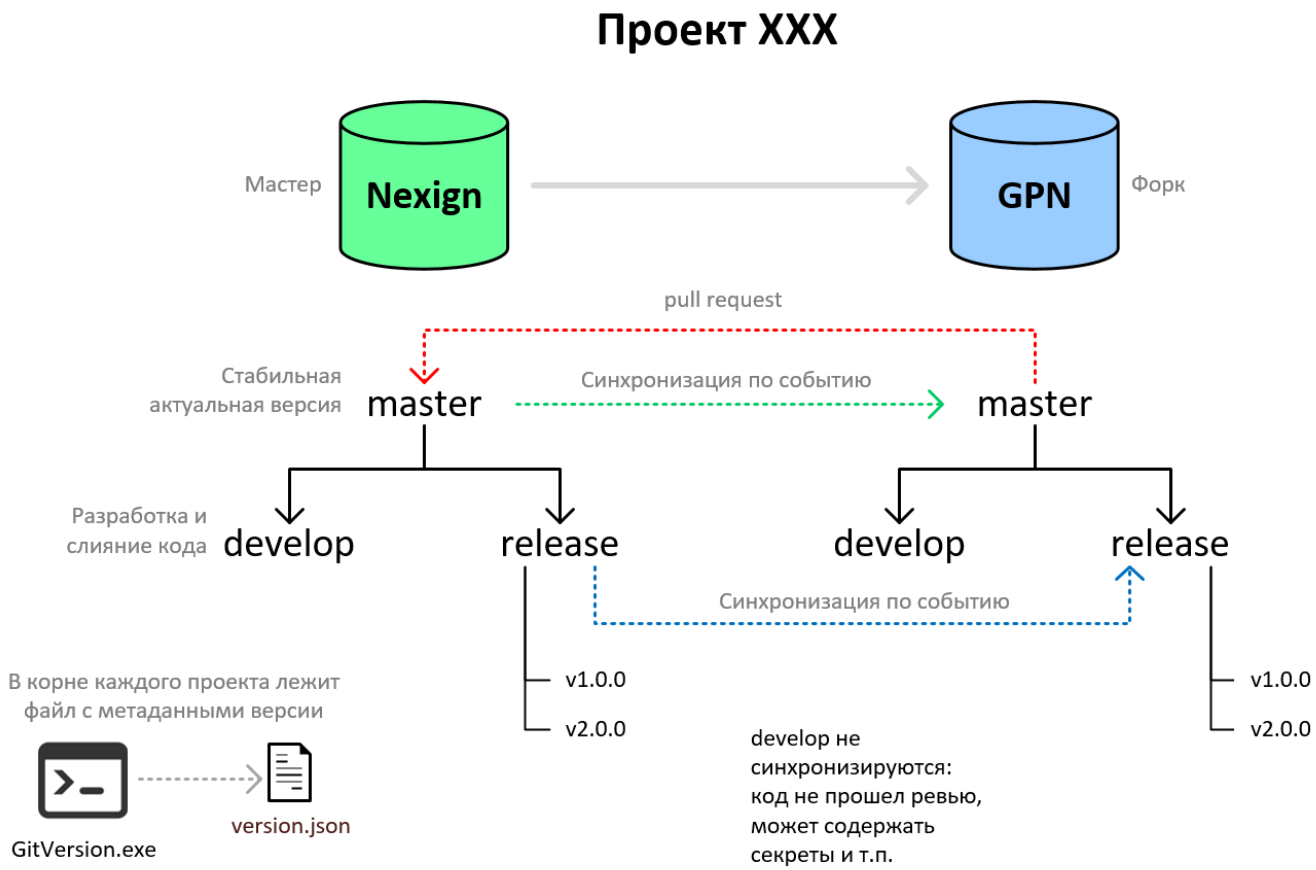
Синхронизация веток

Ветки для синхронизации:

- **master**
- **release/x.y.z**

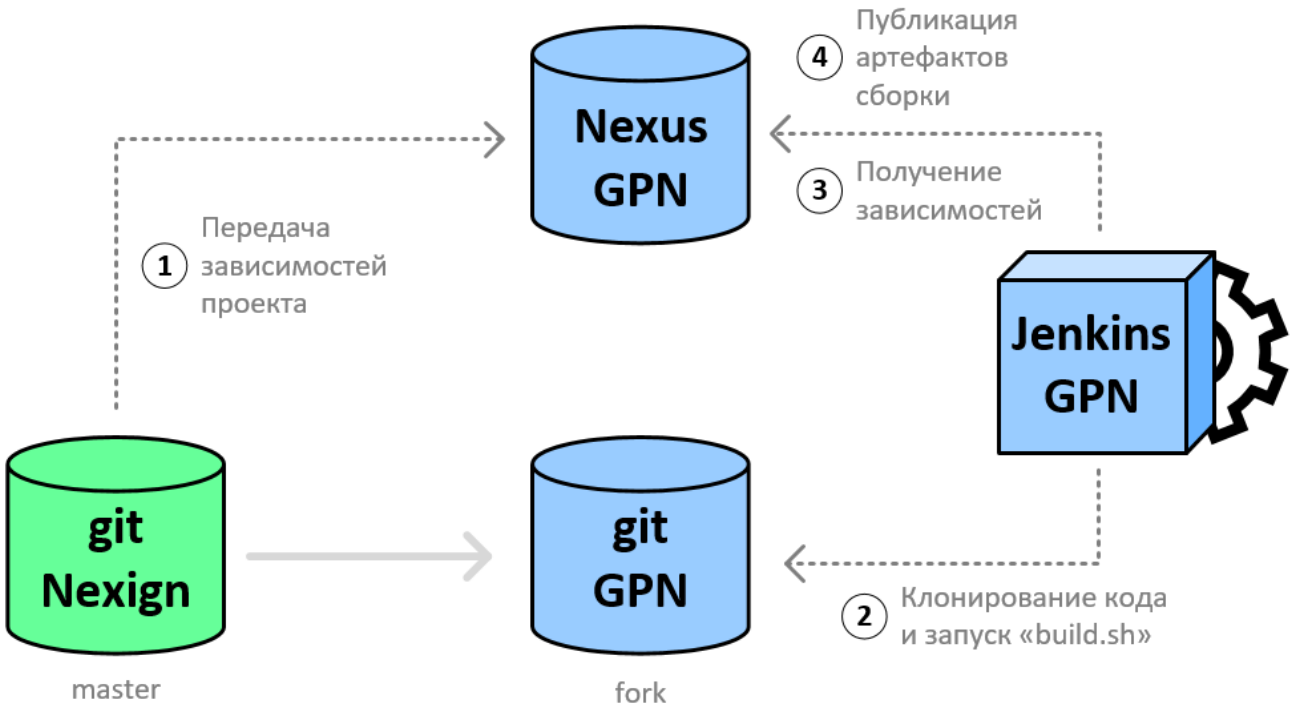
> **Важно**
>
> Ветка **develop** синхронизироваться не будет, так как она может содержать нестабильный код и секреты.

Режим синхронизации не важен и зависит от деталей реализации репозиториев, синхронизация может выполняться по расписанию или по событию коммита (второй вариант удобнее, но сложнее в реализации).



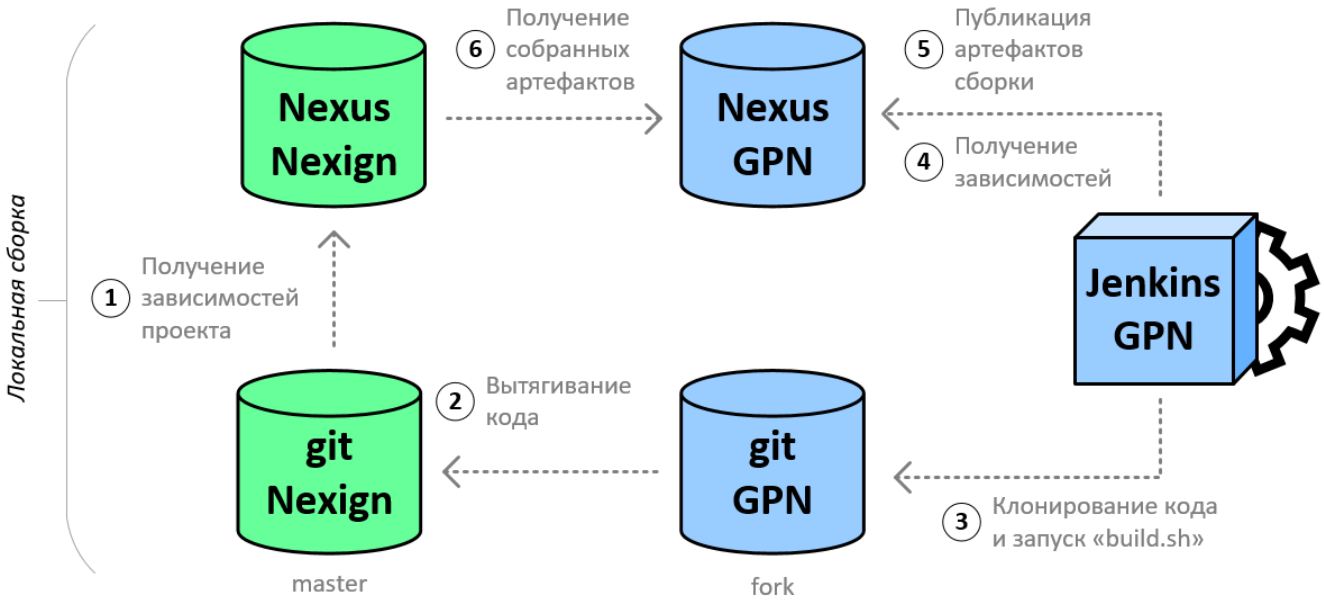
Резервирование зависимостей

Следующая значительная проблема сборки - управление зависимостями. Проект при сборке как правило использует менеджер зависимостей для их скачивания из публичных реестров. Однако, с учетом тех же требований безопасности, сервер Jenkins, на котором выполняется сборка, не может обращаться к внешним ресурсам через интернет. Следовательно, зависимости необходимо каким-то образом доставить в локальный реестр, изолировав таким образом сборку от внешнего мира.



Сценарий доставки зависимостей требует детальной проработки, особенно с учетом проблем, описанных ниже.
В простом случае, когда проекту требуются только стандартные библиотеки из публичных репозиториев с открытым исходным кодом, возможно предварительно предоставить в ГПН список всех зависимостей, и после получения подтверждения, что все зависимости были занесены в Nexus, сборка будет успешной.

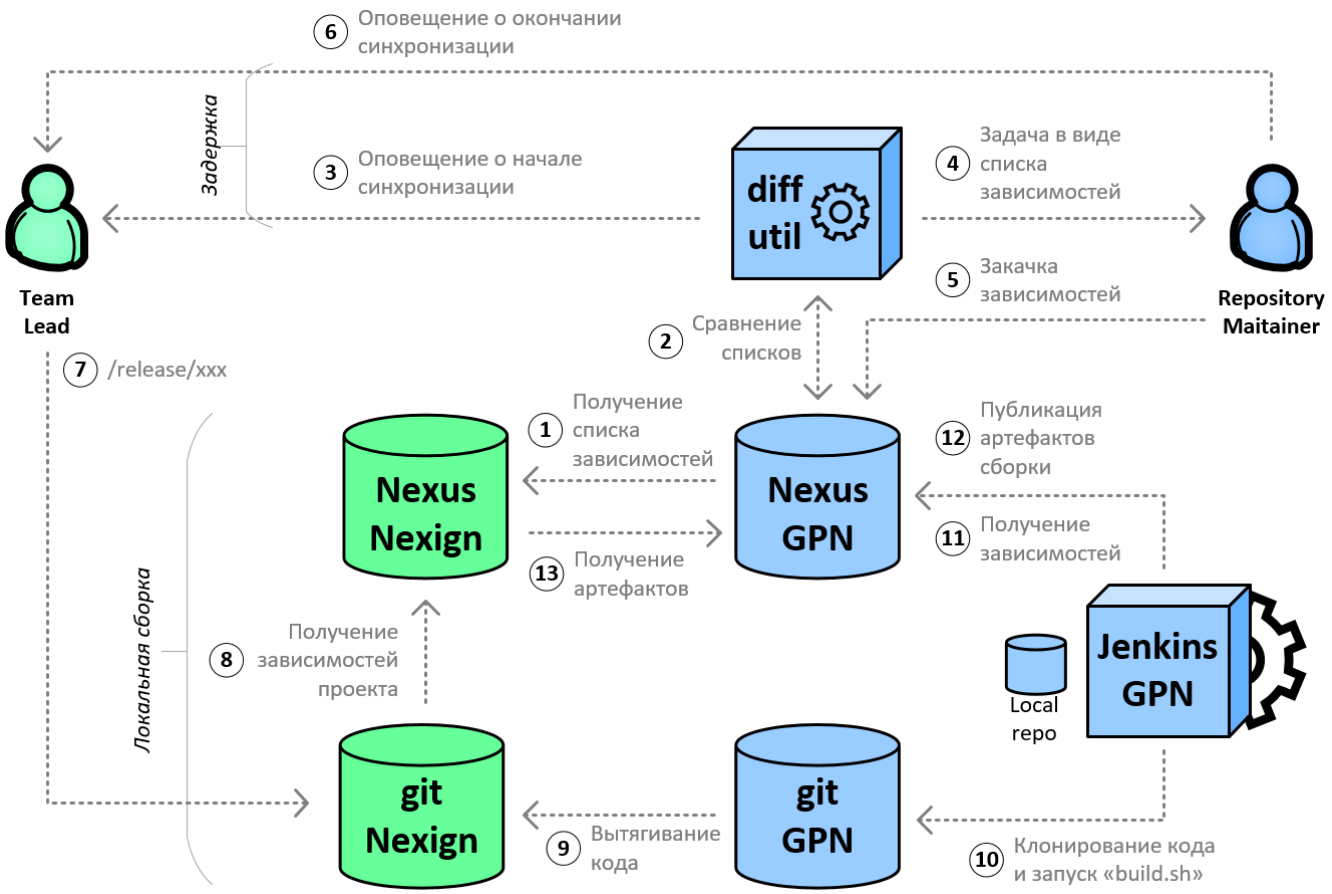
Однако в реальности проекты будут использовать общие компоненты, создаваемые разработчиками отдельно от проектов ГПН. Такой сценарий потребует доработки с обеих сторон: ГПН должна будет предоставить доступ на чтение к реестрам Nexus, а GXT обеспечить одностороннюю синхронизацию реестров таким образом, чтобы совпадали зависимости и их версии.



Следует учесть, что целевая верхнеуровневая архитектура создаваемых систем - микросервисы и контейнеризация. Все это значит, что перечень зависимостей для сборки будет непрерывно расти, причем на всем протяжении проекта. Следовательно, нужно выработать такой механизм, который позволит как можно быстрее выполнять цикл "появление новой зависимости → добавление зависимости в Nexus". На схеме ниже приблизительно описан подобный сценарий.

То есть:

1. По некоторому расписанию специальная утилита diff util запрашивает у Nexus Nexign список всех пакетов.
2. Утилита diff util получает список пакетов из Nexus GPN и ищет различия в этих списках.
3. Отсылается оповещение о начале процедуры синхронизации зависимостей (это важно для планирования работ).
4. Результат сравнения списков пакетов (дельта) отправляется администратору репозитория ГПН.
5. Администратор ГПН закачивает нужные пакеты из Интернет в Nexus GPN.
6. Отсылается оповещение о готовности репозитория к сборке.
7. Тимлид Nexign создает релиз.
8. Выполняется сборка на стороне Nexign, зависимости забираются из Nexus Nexign.
9. Git GPN по событию или расписанию забирает ветку release.
10. Jenkins GPN запускает сбоку (используя как точку входа файл build.sh в корне репозитория)
11. При сборке Jenkins GPN получает зависимости из Nexus GPN.
12. После успешной сборки Jenkins GPN помещает артефакты в Nexus GPN.
13. При оповещении об успешной сборке Nexus Nexign забирает из Nexus GPN нужные ему артефакты.



Профили репозиториев

Для упрощения автоматизации сборки и более быстрого вхождения в проект новых разработчиков следует по возможности стандартизировать структуру репозиториев.

Репозитории под конкретные проекты и на конкретных технологиях должны наследовать общую структуру, различаясь лишь в деталях. На рисунке ниже предлагаемая обобщенная структура репозитория.
Следует рассмотреть более сложный вариант, когда папка tests (а значит, все тесты, кроме модульных) лежат рядом с кодом, но хранятся в отдельном репозитории со своей историей и собственным жизненным циклом.

```
.
 projectXX
     ci                                              #
     docker                                       #  docker   ,       (gradle, npm)
        Dockerfile
     docs                                      #  .  md
     src                                             #
     [tools]                                      #
     [docker-compose.yml]        # docker-compose    . ,
     .gitignore
     CHANGELOG.md
     README.md
     version.json                        #
```

Для единой точки входа сборки используется **`build.sh`.** `Jenkins :`

> **Важно**
>
> Все скрипты сборки (в том числе Jenkins), их настройки и необходимые утилиты должны лежать в папке /ci.

Пример для frontend

```
DOCKER_TAG=%env.DOCKER_REGISTRY%/gpnx/portal/dev/gx-portal-internal:rhel

docker login %env.DOCKER_REGISTRY% -u %env.DOCKER_USERNAME% -p %env.DOCKER_PASSWORD%
docker build -t $DOCKER_TAG . \
--build-arg NPM_CONFIG_REGISTRY=%env.NPM_CONFIG_REGISTRY% \
--build-arg NPM_LOGIN=%npm.auth.login% \
--build-arg NPM_PASSWORD=%npm.auth.password% \
             -f Dockerfile-gpn

[ $? -eq 0 ] || exit 1

 docker push $DOCKER_TAG
```

Файл `version.json` - метаданные версии продукта. Файл может быть изменен вручную либо сгенерирован/изменен специальной утилитой, формирующей версию из метаданных локального репозитория git. В следующих разделах принципы и правила версионирования будут рассмотрены подробнее.

Репозиторий так же должен содержать файлы readme.md и changelog.md.

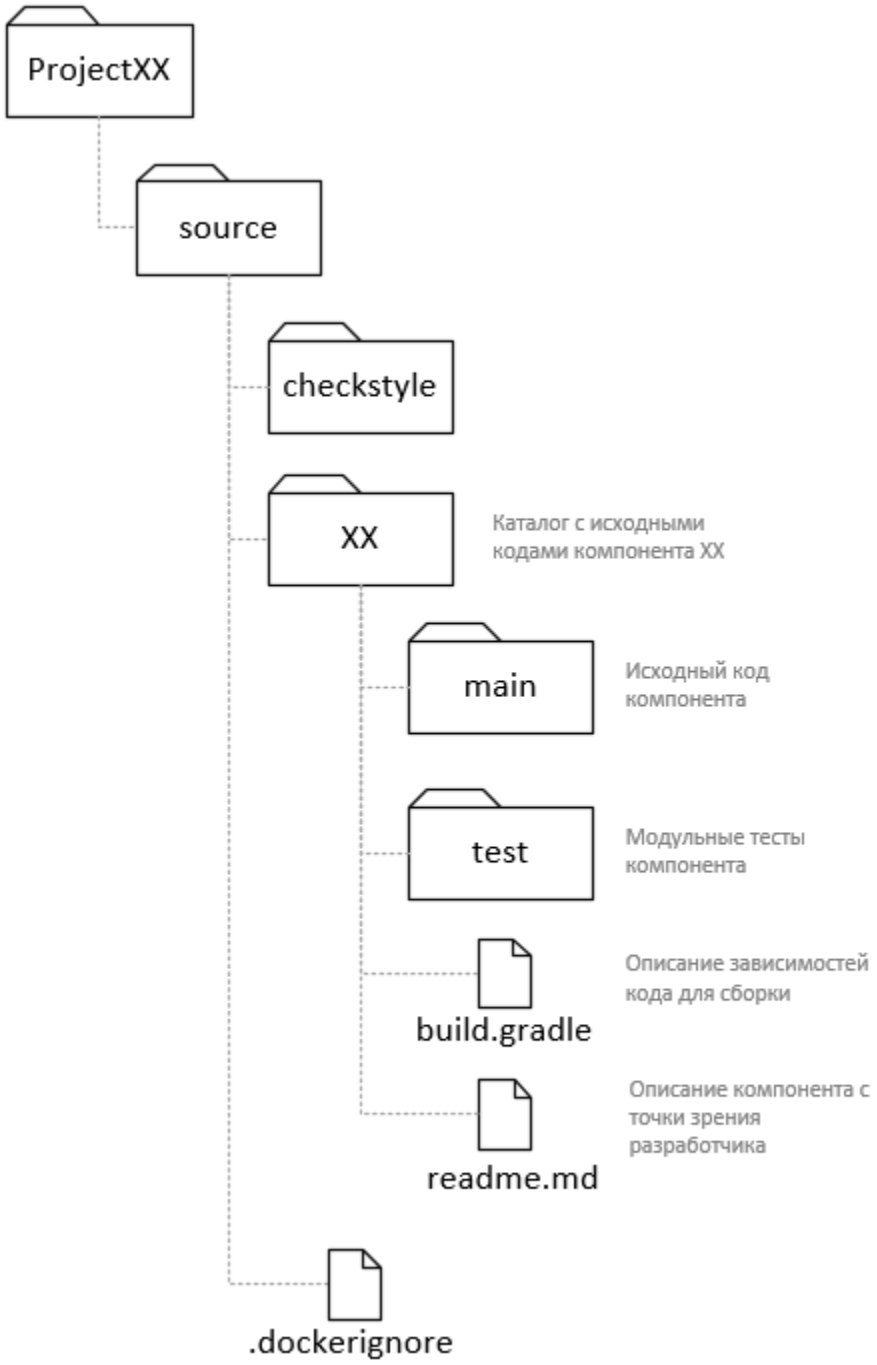
Файл readme.md

Файл readme.md описывает проект и продукт с точки зрения разработчика. В "Приложении 1" описан рекомендуемый формат файла.

Файл changelog.md

Файл changelog.md описывает историю изменений, внесенных в продукт: добавленный функционал, доработки, исправление ошибок. В "Приложении 2" описан рекомендуемый формат с комментариями.

Профиль репозитория для проекта Java



Профиль репозитория для проекта Python

todo

Приложение 1

Рекомендуемый форма файла readme.md

```
# Название продукта

Краткое описание концепции продукта и проекта.

------


## Установка
Как собрать и развернуть продукт?

------

## Использование
Как можно убедиться, что все работает?

-----

## Поддержка
Как оповестить разработчика о проблемах и предложениях?

-----

## План развития
Какие возможности будут у продукта в будущем?

-----

## Разработчики
Кто эти безымянные герои?

-----

## Статус проекта
Только стартовал, в активной работе, завершен?

-----

## Лицензия
Ссылка лицензионное соглашение, если есть.
```

Приложение 2

Рекомендуемый формат файла changelog.md

Changelog
>В этом файле должны быть зарегистрированы все заметные изменения в проекте .

>Формат файла основан на [Keep a Changelog](https://keepachangelog.com/en/1.0.0/),
а версии продукта задаются согласно [Semantic Versioning](https://semver.org/spec/v2.0.0.html).
Основные требования к наполнению следующие:

>- Один подраздел на каждую версию продукта.
>- Релизы перечислены в обратном хронологическом порядке (самые новые – сверху).
>- Все даты указаны в формате YYYY-MM-DD.
>- Заголовок каждого раздела в формате [X.X.X] - YYY-MM-DD, где X.X.X - версия SemVer.
>- Изменения группируются согласно их влиянию на проект следующим образом:
>1. Added для новых функций.
>2. Changed для изменений в существующей функциональности.
>3. Deprecated для функциональности, которая будет удалена в следующих версиях.
>4. Removed для функциональности, которая удалена в этой версии.
>5. Fixed для любых исправлений.
>6. Security для обновлений безопасности.

>- В начале документа всегда должна быть секция Unreleased, чтобы перечислить в ней невыпущенные изменения.

[Unreleased]
Added
- Краткое описание функции

[1.0.0] - 2020-03-01
Added
- Краткое описание функции
- Краткое описание функции
Changed
- Краткое описание изменения функционала
- Краткое описание изменения функционала
Removed
- Краткое описание удаленного функционала

Прикрепленные документы

todo