TĪMEKĻA LIETOTŅU NEPĀRTRAUKTĀ INTEGRĀCIJA UN PIEGĀDE
CONTINUOUS INTEGRATION AND DELIVERY OF WEB APPLICATIONS

Autors: **Aleksandrs BRITUŠKINS**, e-pasts: sansey147@inbox.lv
Zinātniskā darba vadītājs: lektors **Aleksejs ZORINS**
Rēzeknes Tehnoloģiju akadēmija
Atbrīvošanas aleja 115, Rēzekne

Abstract. *With the advent of DevOps practice, rapidly began evolve systems development life cycle shorting methods, such as continuous integration, testing and delivery, release automation and infrastructure as code methods. This methods, is similar to programming scripts which are used to automate a series of static steps to complete defined scenario. Often, the choice of used automation methods and tools depends on project and application complexity, which makes each case unique in its own way. The main goals of the paper are description of basic continuous integration and delivery methods for web based applications with real example using version control system, container virtualization and some auxiliary scripts, which developed by author and currently being successfully used on few real projects. In example used version control system, container virtualization and some auxiliary scripts.*
Keywords: *cloud computing, continuous delivery, continuous integration, DevOps, IT infrastructure.*

Ievads

Aiz jebkura veiksmīga un sarežģīta timekļa projekta visbiežāk pastāv specialistu komandas, kuras nodarbojas ar projekta izstrādi, uzturēšanu un vadību. Eksistē vairākas projektu attīstīšanas metodoloģijas un rīki, toties par vienu no efektīvākiem paņēmieniem mūsdienās var uzskatīt IT procesu automatizāciju. Dotais paņēmiens spēj ietekmēt uz visiem projekta slāņiem, ļaujot gūt kvalitatīvus rezultātus un procesu paātrinājumus.

Rakstā tiks aprakstīts timekļa aplikācijas izstrādes procesa automatizācijas paņēmiens, izmantojot aplikācijas nepārtrauktas integrācijas un piegādes rīkus. Veidojot tā saucamo izstrādes konveijeru tiks izmantota *GitLab* versijas kontroles sistēma, *Docker* konteineru virtualizācija un palīg skripti *Python* programmēšanas valodā. Galvenais mērķis ir paradīt aplikācijas nepārtrauktas integrācijas un piegādes priekšrocības pret statiskām metodēm.

Nepārtrauktas integrācijas un piegādes koncepcijas

Nepārtrauktā integrācija ir programmatūras izstrādes prakse, kuras darbības pamatā ir vairāku koda kopiju apvienošanā vienā galvenajā, lai pildītu uz tās pamata automatizētu montāžu un testus [1]. Dotā prakse ļauj savlaicīgi atklāt potenciālos defektus un integrācijas problēmas, jo liek uzsvaru uz testēšanas automatizāciju, lai pārbaudītu, vai programmatūra nav bojāta ikreiz, kad galvenajā koda versijā tiek pievienota jauna versija. Pāreja uz nepārtrauktu integrāciju samazina integrācijas sarežģītību un padara to paredzamāku, savlaicīgi atklājot un novēršot kļūdas un pretrunas. Visbiežāk nepārtrauktas integrācijas ieviešanai izmantojas versijas kontroles sistēmas.

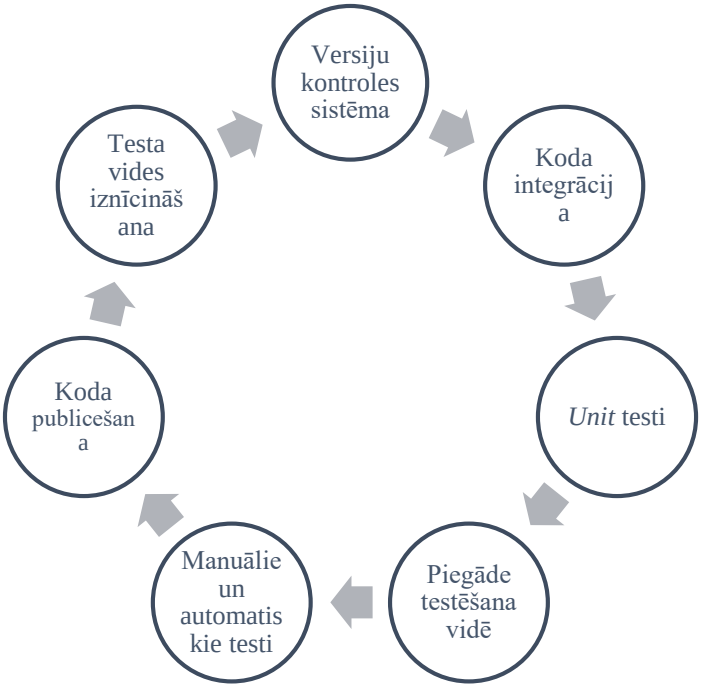
Nepārtraukta piegāde ir programatūras izstrādes prakse, kurā vasas jaunas funkcijas, konfigurācijas izmaiņas, kļūdu labojumi un eksperimenti, tiek piegādāti testa vai produkcijas vidē pēc iespējas ātrāk un drošāk, ideālā gadījumā ar minimāliem vai bez globālas ietekmes uz produkta darbību [1].

Pēc būtības nepārtraukta piegāde ir nepārtrauktas integrācijas paplašinājums un visu procesu var attēlot šādi (skatīt 1. attēlu):

1. Izstrādātājs nosuta koda izmaiņas versiju kontroles sistēmā
2. Integrēšanas serverī uzsākas konkrēta koda versijas integrēšanas process.
3. Uzsāk darbību *unit* testi, ja tādi tika izveidoti un integrēti projektā.
4. Integrēta programmatūras versija tiek piegādāta testēšanas serverī, kur tiek uzsākts nākamais testēšanas posms, piemēram manuālais.

http://dx.doi.org/10.17770/het2019.23.4395

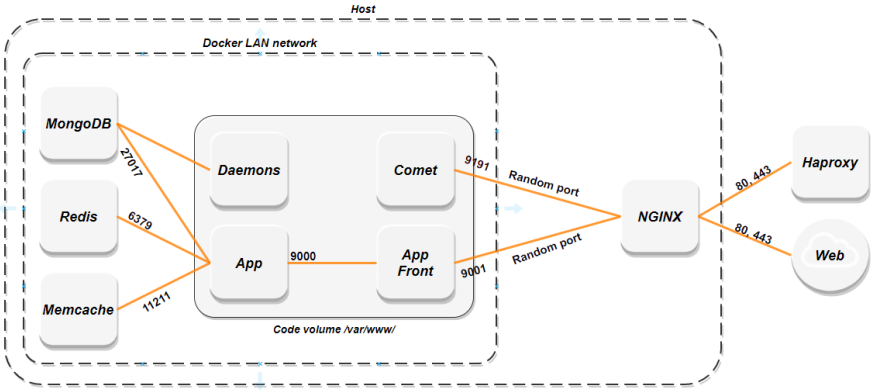
5. Pēc visu iepriekšējo soļu veiksmīgās iziešanas, projekts saņem notestētu programmatūras kodu kurš ir gatavs integrēšanai galvenajā versijā.
6. Iepriekšējā testa vide tiek iznīcināta, resursu atbrīvošanai.



1.attēls. Nepārtrauktas integrācijas un piegādes shēma

Pielietojamo eksperimentā rīku apraksts

Par projekta pamatu tiek paņemts PHP projekts ar dažādiem papildus mikro servisiem, bet dotajā gadījumā tā detalizēts apraksts nav vajadzīgs, svarīgs ir tikai tā abstrakts tēls un uzbūves sastāvdaļas (skatīt 2. attēlu), priekš nepārtrauktas integrācijas un piegādes aprakstīšanas. Papildus servisi ir *nginx http* serveris, *mongodb* datu bāze, *redis* rindu brokeris, *memcached* sīkdatņu datubāze un divi *php* mikro servisi (*daemons* un *comet*).



2.attēls. Aplikācijas sastāvdaļu shēma

Gitlab ir plaši izplatāmā un attīstīta versiju kontroles sistēma, kuras ietvaros ir daudzfunkcionālā nepārtrauktas integrācijas un piegādēs funkcionalitāte. Eksperimenta ietvaros tiks izmantots *gitlab runner* agents kurš tiek instalēts uz testa servera lai sistēma spētu mijiedarboties ar to. Aģenta konfigurācijas fails jeb darbības scenārijs tiek rakstīts uz *YAML* datu attēlošanas valodas [2].

Docker konteineru virtualizācija ļauj "iepakot" visu aplikācijas apkārtni ar tas sastāvdaļām un atkarībām iezolētajā vidē, kuru tālāk var pārnest un iedarbināt gandrīz jebkurā

sistēmā. *Docker* pamatā izmantojas tēli kuri tiek veidoti ar slāņu palīdzību [3]. Konfigurācijas arī tiek rakstītas uz *YAML* valodas. Dota tehnoloģija ļauj reizēs atvieglot vairākus automatizācijas procesus kā arī gala izmaksas uz tām. Tā kā projekts sastāv no vairākiem servisiem eksperimenta daļā tiks lietota *docker* kompozīcija, kura pēc būtības ir vairāku tēlu apvienojums.

Debian timekļa serveris uz kura tiks piegādāta un piedarbināta integrētā aplikācija.

Eksperimentālā daļa

Uzsākt ir vērts ar *docker* tēla konfigurācijas (skatīt 1. tabulu) izveidošanu priekš aplikācijas, kurš tiek veidots ar YAML valodas palīdzību un konfigurācijas fails tiek izvietots projekta pamat mapē [4].

1. tabula

***Docker* aplikācijas tēla konfigurācijas slāņu tabula**

Slānis	Komanda	Apraksts
1.	*FROM* php:7.2-fpm	tiek definēts pamata tēls
2.	*RUN* apt-get update && apt-get install -y \ apt-transport-https apt-utils libmemcached-dev build-essential curl software-properties-common zlib1g-dev git libicu-dev python3-pip python3-dev	aplikācijas atkarību pakešu instalāciju
3.	*RUN* docker-php-ext-install opcache mbstring zip bcmath intl pcntl sockets && pecl install mongodb \ memcached && docker-php-ext-enable memcached \ mongodb	aplikācijas atkarību pakešu instalāciju
4.	*RUN* curl –sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin - filename=composer	aplikācijas atkarību pakešu instalāciju
5.	*COPY* . /var/www/	kods no tekošās mapes tiek iekopēts iekš konteinera
6.	*WORKDIR* /var/www/	tiek pāriets uz koda mapi iekš konteinera
7.	*RUN* pip3 install ruamel.yaml && python3 build/scripts/configure.py "$env"	tiek pielietots papildu skripts kurš pielieto dinamiskās konfigurācijas aplikācijai
8.	*USER* www-data	lietotāja maiņa
9.	RUN php -d memory_limit=-1 /usr/local/bin/composer update --ignore-platform-reqs	*sympfony framework* atkarību instalācija
10.	*EXPOSE* 9000	tiek atvērts konteinera ports
11.	*CMD* ["php-fpm"]	aplikācijas darbināšana

Tālāk tika izveidots *docker* kompozīcijas konfigurācijas fails (skatīt 3. attēlu), kurš arī tiek izvietots projekta pamat mapē. Konfigurācijā tika pievienoti papildus tēli ar kuriem mijiedarbojas esošā aplikācija. Tie ir *nginx http* serveris, *mongodb* datu bāze, *redis* rindu brokeris, *memcached* sīkdatņu datubāze un divi *php* mikro servisi (*daemons* un *comet*).

```
version: '3.1'
services:
  app:
    build:
      context: .
    restart: always
    depends_on:
      - daemons
      - mongo
      - memcached
      - redis
      - rabbitmq
    volumes:
    - code:/var/www
  app-front:
    build: build/nginx
    restart: always
    depends_on:
      - app
      - comet
    ports:
      - '9001'
    volumes:
      - code:/var/www
  comet:
    build: build/comet
    restart: always
    ports:
      - '9191'
    volumes:
      - code:/var/www
  mongo:
    build: build/mongodb
    restart: always
  memcached:
    build: build/memcached
    restart: always
  redis:
    build: build/redis
    restart: always
  daemons:
    build: build/daemons
    restart: always
    volumes:
      - code:/var/www
volumes:
  code:
```

3.attēls. *Docker* kompozīcijas konfigurācijas fails

Tā kā par nepārtrauktas integrācijas un piegādes riku tiek paņemts *gitlab* versiju kontroles sistēma, tam arī tiek veidots konfigurācijas fails (skatīt 4.-9. attēlu), kurā tiek aprakstītas nepieciešamās komandas *YAML* valodā [5]. Konfigurācijas faila sākumā tiek definēti trīs uzdevumu veidi (skatīt 4. attēlu).

```
stages:
  - build
  - deploy
  - stop
```

4.attēls. *Gitlab runner* uzdevumu definēšana

Nākamajā solī tiek definēti divi uzdevumi kuri atkārtosies citos uzdevumos, tāpēc tie tiek iznesti kā mainīgie.

Pirmais atkārtošanas uzdevums (skatīt 5. attēlu), tiek definēts kā sagatavošanas posms priekš nepārtrauktas piegādes un veic pagājušo konfigurāciju un konteineru tīrīšanu, ja tādi eksistē.

```
.preparing_deploy: &preparing_deploy_job
  before_script:
    # Remove front configurations
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.conf || true
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.payments.conf|| true
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.games.conf|| true
    # Change working directory
    - cd $MAIN_FOLDER/$CI_COMMIT_REF_NAME/$CI_PROJECT_NAME
    # Stop and remove old containers
    - docker-compose -p $CI_COMMIT_REF_NAME down -v || true
```

5.attēls. *Gitlab runner* sagatavošanas uzdevums

Otrais atkārtošanas uzdevums (skatīt 6. attēlu), tiek definēts kā dinamiskais *http* servera konfigurāciju failu ģenerators, kurš izmantojot esošos faila paraugus un *gitlab* repozitorija mainīgos veido ar skripta palīdzību dinamisko timekļa konfigurāciju konkrētajai koda versijai.

```
.configure_front: &configure_front_job
  after_script:
    # Create nginx configuration file
    - cp build/nginx/configs/front.conf $NGINX_FOLDER/
$CI_COMMIT_REF_NAME.conf
    # Create payments configuration file
    - cp build/nginx/configs/payments.conf $NGINX_FOLDER/
$CI_COMMIT_REF_NAME.payments.conf
    # Create games configuration file
    - cp build/nginx/configs/games.conf $NGINX_FOLDER/
$CI_COMMIT_REF_NAME.games.conf
    # Configure nginx host
    - cd /data/scripts/ && python configure_front.py
$CI_COMMIT_REF_NAME $CI_PROJECT_NAME $NGINX_FOLDER
    # Reload nginx
    - nginx -s reload
```

6.attēls. *Http* servera konfigurāciju ģeneratora uzdevums

Tālāk tiek definēts integrācijas un piegādes uzdevums (skatīt 7. attēlu), kas vispirms pārkopē kodu no versijas kontroles sistēmas uz serveri, pēc tam uzģenerē ar skripta palīdzību dinamiskos mainīgos un iebūvē tos projekta konfigurācijas failos. Tālāk tiek veidoti timekļa aplikācijas tēlus, balstoties uz iepriekš definētiem konteinerizācijas konfigurācijām.

```
build:
  stage: build
  before_script:
    # Remove old project directory
    - rm -rf $MAIN_FOLDER/$CI_COMMIT_REF_NAME || true
    # Create new project directory
    - mkdir $MAIN_FOLDER/$CI_COMMIT_REF_NAME
    # Clone branch code
    - cd $MAIN_FOLDER/$CI_COMMIT_REF_NAME/ && git clone -b
$CI_COMMIT_REF_NAME --single-branch $CI_REPOSITORY_URL
    # Copy cache if exist
    - docker cp -L ${CI_COMMIT_REF_NAME}_app_1:/var/www/.composer
$MAIN_FOLDER/$CI_COMMIT_REF_NAME/$CI_PROJECT_NAME/ || true
    - docker cp -L ${CI_COMMIT_REF_NAME}_app_
1:/var/www/node_modules $MAIN_FOLDER/$CI_COMMIT_REF_NAME/
$CI_PROJECT_NAME/ || true
    # Add pay/game keys in configuration
    - cd /data/scripts/ && python3 configure.py
$CI_COMMIT_REF_NAME $CI_PROJECT_NAME
  script:
    # Build project images
    - cd $MAIN_FOLDER/$CI_COMMIT_REF_NAME/$CI_PROJECT_NAME
    - docker-compose -p $CI_COMMIT_REF_NAME build
    - docker-compose -p $CI_COMMIT_REF_NAME build --build-arg
slug=$CI_COMMIT_REF_NAME
  when: manual
  retry: 2
```

7.attēls. *Gitlab runner* nepārtrauktās integrācijas un piegādes uzdevums

Projekta iniciācijas uzdevumā (skatīt 8. attēlu) tiek pielietoti iepriekš definētie atkārtošanas uzdevumi, pēc kuriem tiek palaisti projekta konteineri un projekta inicializācijas etaps.

```
deploy:
  stage: deploy
  <<: *preparing_deploy_job
  script:
    # Start new app
    - docker-compose -p $CI_COMMIT_REF_NAME up -d app
    # Run composer deploy
    - docker-compose -p $CI_COMMIT_REF_NAME exec -T app
/usr/local/bin/composer deployFull
    # Run front app
    - docker-compose -p $CI_COMMIT_REF_NAME up -d app-front
  <<: *configure_front_job
  when: manual
  retry: 2
  environment:
    name: $CI_COMMIT_REF_NAME
    url: https://$CI_COMMIT_REF_NAME.$HOST
    on_stop: stop
```

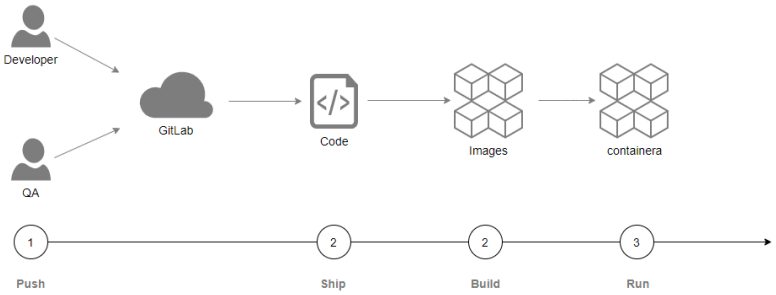
8.attēls. *Gitlab runner* inicializācijas uzdevums

Pēdējais uzdevums (skatīt 9 attēlu), atbild par projekta koda, konfigurācijas failu un konteineru iznicināšanu un dzēšanu no testēšanas servera.

```
stop:
  stage: stop
  before_script:
    # Change working directory
    - cd $MAIN_FOLDER/$CI_COMMIT_REF_NAME/$CI_PROJECT_NAME
  script:
    # Stop and remove project images/containers
    - docker-compose -p $CI_COMMIT_REF_NAME down -v --rmi all --
remove-orphans || true
  after_script:
    # Remove project folder
    - rm -rf $MAIN_FOLDER/$CI_COMMIT_REF_NAME || true
    # Remove front configuration
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.conf || true
    # Remove payments configuration file
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.payments.conf || true
    # Remove games configuration file
    - rm $NGINX_FOLDER/$CI_COMMIT_REF_NAME.games.conf || true
  when: manual
  environment:
    name: $CI_COMMIT_REF_NAME
    action: stop
```

9.attēls. *Gitlab runner* iznicināšanas uzdevums

Rezultātā tika izveidots sekojošais timekļa aplikācijas process (skatīt 10. attēlu) – izstrādātājs publicē projekta kodu atsevišķajā versijā un nodod ziņu testētājam ka var uzsākt testēšanu. Testētajs savukārt nospiežot pogu versijas kontroles sistēmā palaiž koda integraciju, pēc kura kods tiek piegādāts uz testa servera un uzsāk iebūvēt to konteneru tēlos. Tālāk testētajs palaiž kontenerus un tiek izveidota atsevišķa projekta vide zem uzģenerētās timekļa adreses, kur arī uzsākas testi. Viss šīs proces aizņem ne vairāk par 5 minūtēm. Pēc testu pabeigsanas ar testētāja komandu, testēšanas vide tiek iznīcināta.



10.attēls. Iestrādātā nepārtrauktās integrācijas un piegādes procesa shēma

Summary

The paper will describe automation technique of development process, using the tools for continuous integration and continuous delivery. The so-called development conveyor will use the GitLab version control system, Docker container virtualization, and help scripting in Python programming language. The main goal is to show the benefits of continuous application integration and delivery against static methods.

First, we need to find out what is continuous integration, continuous delivery and how these methods can help in application development process. So continuous integration is a software development practice based on combining multiple code copies into a single master to perform automated assembly and testing based on it. This practice allows you to detect potential defects and integration problems in a timely manner, as it puts emphasis on testing automation to check if the software damaged every time a new version added to the main code version. The transition to continuous integration reduces the complexity of integration and makes it more predictable by timely detection and elimination of errors and contradictions. Most often, version control systems used to implement continuous integration.

In addition, continuous delivery is a software development practice where new features, configuration changes, bug fixes and experiments delivered to the test or production

environment as quickly and safely as possible, ideally with minimal or no global impact on product performance.

In the developed experimental part, the continuous integration and delivery method of the web application introduces the project as an opportunity to test the new functionality in the production environment, to test external communication, to reduce costs to final testing and the implementation process. It is worth mentioning that the method is not final, as the development of the project requires the modification of continuous integration and delivery configurations to actual project situation.

Secinājumi

Izstrādātā eksperimentālajā daļā timekļa aplikācijas nepatrauktas integrācijas un piegādes metode ieviešprojektam tādas priekšrocības kā iespēju testēt jauno funkcionalitāti vidē maksimāli pietuvinātai produkcijai, testēt ārējos sakarus, samazināt izmaksas uz galīgo testēšanu un ieviešanas procesu. No trūkumiem ir vērts pieminēt ka metode nav galīgā, jo tika izstrādāta un piemērota konkrēta projekta darbībai, kā arī attīstoties projektam, jāmodificē arī nepārtrauktās integrācijas un piegādes konfigurācijas līdz aktuālajiem projekta stāvokļiem.

Tika izskatīta nepārtrauktās integrācijas un piegādes būtne, izmantojot kuru tika panākta strādājošā piemēra izveidošana, kas pierāda ka ieviešot doto metodi, izstrādes projekts iegūst vairākus ieguvumus, toties lai pilnvērtīgi ieviest un uzturēt metodi un procesu, ir nepieciešams līgt specialistus.

Literatūra

1. Armstrong S. *DevOps for Networking.* Packt Publishing Ltd, 2016
2. Duffy M. *DevOps Automation Cookbook*, Packt Publishing Ltd, 2015
3. Swartout Paul *Continuous Delivery and DevOps – A Quickstart Guide, 2nd* Edition, Packt Publishing Ltd, 2014
4. Soni M. - *DevOps for Web Development, Packt Publishing Ltd,* 2016
5. Verona J. - *Practical DevOps,* Packt Publishing Ltd, 2016