DevOps Projects

- Intégration continue dans un environnement cloud avec Gitea
- Boissipay

Intégration continue dans un environnement cloud avec Gitea

Introduction

image-20220306133642473.png

Slides de Présentation

L'objectif de ce projet était de monter un stack d'intégration continue sur une infrastructure cloud.

L'idée de base était de "mettre en place un serveur git".

C'était la première idée sélectionnée., une fois ce serveur en place et fonctionnel, nous avons pu rajouter des services par-dessus.

Ainsi, beaucoup de services d'intégration continue viennent se greffer sur note serveur Gitea. L'objectif était déjà d'avoir notre propre ecosystem d'intégration /déploiement continue, en ayant la main sur tous les services présents.

Les applications qui sont finalement mises en place :

- Gitea
- SonarQube
- Drone CI
- Docker registry

Les systèmes d'hébergement qui ont été utilisés :

- AWS
- DigitalOcean
- ScaleWay
- PulseHeberg

Les services mis en place :

- AWS Elastic File System
- AWS Relational Database Service
- AWS Elastic Beanstalk with Docker
- AWS Route 53
- ScaleWay Simple Service Storage
- ScaleWay Docker Registry
- Digital Ocean Droplet with Docker (SonarQube)
- PulseHeberg Instance with Docker (Drone CI)

Accès aux services

- Gitea => https://gitea.nospy.fr/
- SonarQube => https://sonarqube.les-evades-du-chenil.dog/
- Drone CI => https://drone.nospy.fr/
- Application en déploiement continue (pokefight) => https://pokemon.nospy.fr/

Découpage des applicatifs

Gitea

La stack gitea est majoritairement hébergé sur AWS.

L'instance Gitea est lancé dans un conteneur docker.

Nous utilisons des **instances ec2** lancées via **elastic beanstalk** avec des instructions spécifiques pour pouvoir exploiter Docker.

Un load balancer combiné avec route 53 (service AWS de gestion des noms de domaines, DNS et certificats SSL) nous permet d’accéder à l'application via le web avec une URL défini et protégé par un certificat SSL.

Les volumes de gitea sont monté sur **E.F.S. elastic file system** service de gestion de fichiers d'Amazon.

Les gros fichiers comme les archives, les images ... gitea proposent d'utiliser un service basé sur le protocole S3 pour stocker ses fichiers. Nous utilisons cette solution en combinaison avec le système S3 object storage de **Scaleway**.

Pour la base de données gitea propose plusieurs solutions. La solution que nous avons sélectionnée est **PostgreSQL**, qui fonctionne grâce à **AWS RDS**.

La stack gitea est majoritairement hébergé sur AWS

Fonctionnement

Gitea fonctionne comme les autres serveurs git, il permet même d'importer des projets de ses concurrents.

Cela nous a permis d'alimenter rapidement notre serveur avec des dépôts de projets personnels qui se prête bien à l'intégration continue

Quand on ouvre un projet, on arrive sur cette interface :

gitea-project-view.png

Chaque utilisateur a aussi accès à un fil d'actualité des projets auquel il se réfère.

gitea-dashboard.png

Docker Registry

Pour héberger notre registre docker, nous utilisons un service ches Scaleway.

Nous avons fait ce choix, car le tier gratuit assez généreux en termes de stockage. On se sert de ce registre pour stocker les images de nos applications qui ont pu être générés via des intégrations continues sur certains dépôts.

On s'en sert également pour faire du déploiement continu, grâce à la combinaison de **Drone CI** et **PulseHeberg**.

image-20220129155359004.png

image-20220129155508836.png

Drone CI

Drone CI est un outil d'intégration continue qui vient se plugger sur Gitea ou autre serveur git.

Il permet de définir des steps pour organiser des opérations de lancement de build, de test, de déploiement ou d'appel à des services externes (SonarQube dans notre exemple).

Pour les détails à propos du DNS et de la configuration SSL c'est quelque chose qui est géré par **caddy**.

Drone CI est séparé en deux parties :

- Un applicatif web (Drone CI)
- Des runners qui permettent l'exécution des étapes. Dans notre cas, nous avons 2 runners.

    - Un présent sur le même serveur que l'applicatif web, et qui couvre quasiment tous les cas d'usages
    - Un présent sur un serveur personnel, plus puissant, et qui permet d'exécuter des tâches plus lourdes que le premier ne peut pas faire (ex : des build d'applications Angular qui sont très consommateur en mémoire (+2Go)).

Exemple d'utilisation de Drone :

drone-build-schema.png

drone-build-screen.png

Sur ces images on voit le déroulement d'un build en 5 étapes :

1. Clonage du projet Git
2. Création et envoie d'un rapport à SonarQube
3. Lancement des tests unitaires
4. Lancement du build de l'image docker et push sur le registre Docker (**ScaleWay**)
5. Déploiement de la nouvelle image sur un serveur **PulseHeberg**

Cela nous permet d'avoir le déroulement suivant :

image-20220206162429080.png

Interaction avec Gitea

Lorsque l'on veut initialiser un dépôt dans drone, ce dernier va en réalité créer un Webhook dans Gitea pour qu'il soit prévenu en cas de nouveau commit pour lancer un build.

image-20220129160310394.png

Fichier de configuration

La configuration d'un build Drone se fait dans un fichier `drone.yml`.

```
kind: pipeline
name: build image and push to registry
type: docker

steps:
  - name: install dependencies
    image: node:14
    commands:
      - npm ci
  - name: generate coverage report
    image: node:14
```

```
    failure: ignore
    commands:
      - npm test
  - name: SonarQube
    image: aosapps/drone-sonar-plugin
    settings:
      sonar_host: https://sonarqube.les-evades-du-chenil.dog
      sonar_token:
        from_secret: SONAR_TOKEN
  - name: test
    image: node:14
    commands:
      - npm test
  - name: build & push
    image: plugins/docker
    settings:
      username: nologin
      password:
        from_secret: REGISTRY_TOKEN
      repo: rg.fr-par.scw.cloud/gitea/pokefight
      registry: rg.fr-par.scw.cloud/gitea
  - name: deploy new image
    image: plugins/webhook
    settings:
      urls:
        from_secret: SERVICE_WEBHOOK
      method: POST
      debug: true


node:
  location: home
```

La définition des secrets se fait dans drone sur cette interface :

drone-secret.png

SonarQube

SonarQube est un outil qui permet d'inspecter la qualité du code, sa sécurité, la couverture des tests, ...

L'applicatif SonarQube est hébergé sur **un droplet** chez **DigitalOcean**. Il fonctionne lui aussi sur une stack docker avec un **PostgreSQL** local.

Pour les détails à propos du DNS et de la configuration SSL c'est quelque chose qui est géré par **caddy**.

`docker-compose.yml`

```
version: "3"

services:
  sonarqube:
    image: sonarqube:community
    depends_on:
      - db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: xxxx
      SONAR_JDBC_PASSWORD: xxxx
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
    ports:
      - "9122:9000"
  db:
    image: postgres:12
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

volumes:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:
```

sonarqube-HomePage.png

Configuration des rapports de couverture de tests

Dans un projet Java, il suffit d'installer la dépendance **Jacoco**, qui créé un rapport de couverture de test exploitable par **SonarQube**.

Dans le cas d'un projet sonar-project.properties, il faut configurer quelques options supplémentaires (demander à Jest de générer un rapport, installer une dépendance spéciale de sonar, **jest-sonar-reporter** pour avoir un rapport interprétable) et créer un fichier de configuration **sonar-project.properties** pour l'aider à identifier tout ça :

```
sonar.language=ts
sonar.sources=src
sonar.tests=src
sonar.testExecutionReportPaths=test-report.xml
sonar.javascript.lcov.reportPaths=coverage/lcov.info
sonar.inclusions=src/**/*.ts
sonar.exclusions=**/node_modules/**,**/*.spec.ts,**/*.test.ts
sonar.test.inclusions=**/*.spec.ts,**/*.test.ts
```

S3

S3 (Simple Service Storage) est à l'origine un service d'AWS qui propose du stockage d'objet à travers le web. C'est un produit qui a été très populaire et beaucoup de concurrents ont créé des API qui fonctionnent de la même manière et qui sont compatibles avec l'API S3 d'AWS.

C'est le cas de **ScaleWay**, nous avons donc choisi de mettre en place le service de stockage des fichiers volumineux et des archives sur un stockage S3 chez eux afin d'alléger la charge du stockage local chez AWS (EFS).

C'est quelque chose de configurable dans les paramètres de Gitea que de pouvoir déplacer le stockage des fichiers volumineux, qui sont appelés depuis le navigateur de l'utilisateur sur un bucket S3.

Détail étape par étape

La première brique que nous avons mise en place est le Gitea sur AWS.

Nous avons commencé par mettre en place les différents modules chacun de notre côté (RDS et S3).

Ensuite, nous nous sommes réunis pour mettre en place l'instance Gitea.

Nous avons fait une tentative avec ECS (Elastic Container Service) mais on s'est très vite rendu compte que c'était hors du tier gratuit. On s'est donc ravisé et on s'est servis d'Elastic Beanstalk pour faire tourner l'image docker de Gitea sur une instance EC2.

gitea-instance-aws.png

---

gitea-instance-ip-aws.png

L'un des passages obligé d'AWS, c'est les groupes de sécurité et les VPC. Dans notre cas, nous avons un VPC qui contient l'instance AWS. Nous mettons des images de la configuration des VPC et des groupes de sécurité en annexe, car il y en a beaucoup :

- Un VPC qui contient "tout"
- Un groupe de sécurité pour le load balancer
- Un groupe de sécurité pour l'instance EC2 (Elastic Beanstalk)
- Un groupe de sécurité pour la base de donnée (PostgreSQL via RDS)
- Un groupe de sécurité pour l'EFS (le système de fichier qui accueil les volumes de gitea)

Voici quelques configurations importantes :

La configuration du VPC qui contient tout l'écosystème

vpc-aws.png

La configuration de la base de donnée sur Amazon RDS

configuration-rds.png

La configuration DNS (le domaine a été acheté sur ovh)

configuration-route53.png

Une fois ceci mis en place, nous avons cherché des utilitaires qui se greffait sur Gitea.

Nous savions que nous recherchions des outils d'analyse de code et un outil de build.

Pour l'outil d'analyse de code, on a fait au plus classique, car l'un des rares open source est SonarQube.

On héberge cette instance de SonarQube sur **Digital Ocean**. L'un d'entre nous possédait déjà un serveur SonarQube, mais nous voulions un service intégré au projet.

Pour l'outil de build par contre, nous nous sommes aperçu que Gitea s'interfaçait bien avec des "petits" outils de build. Premièrement, nous avons regardé **Agola** qui est un petit outil de build. Cependant, il demande de faire appel à un service externe pour l'authentification. Ensuite, nous avons regardé du côté de Drone CI et l'outil nous a conquis, à la fois simple et puissant avec une

belle interface graphique.

Une fois Drone CI en place, on a décidé de mettre en place un registre Docker pour pouvoir stocker les images construites avec Drone CI.

Drone CI est aussi lié à SonarQube, c'est-à-dire qu'il lance un scan sur les commits et lui transmet le rapport.

Annexe

Configuration des groupes de sécurités :

Load Balancer

gs-load-balancer-1.png

gs-load-balancer-2.png

gs-load-balancer-3.png

EC2

gs-ec2-1.png

gs-ec2-2.png

gs-ec2-3.png

RDS

gs-rds-1.png

gs-rds-2.png

gs-rds-3.png

EFS

gs-efs-1.png

gs-efs-2.png

gs-efs-3.png

Configuration Elastic Beanstalk

Architecture du dossier

image-20220129162754185.png

Lancement Docker

`Dockerrun.aws.json`

```
{
  "AWSEBDockerrunVersion": "1",
  "Image": {
    "Name": "gitea/gitea:latest",
    "Update": "true"
  },
  "Ports": [
    {
      "ContainerPort": 3000,
      "HostPort": 3000
    }
  ],
  "Volumes": [
    {
      "HostDirectory": "/gitea/data",
      "ContainerDirectory": "/data"
    }
  ]
}
```

.elasticbeanstalk

`config.yml`

```
branch-defaults:
  main:
    environment: giteaesgi-env
global:
  application_name: gitea-esgi
```

```
  branch: main
  default_ec2_keyname: null
  default_platform: Docker running on 64bit Amazon Linux 2
  default_region: eu-central-1
  include_git_submodules: true
  instance_profile: null
  platform_name: null
  platform_version: null
  profile: eb-cli
  repository: null
  sc: git
  workspace_type: Application
```

.ebextensions

`alb-http-to-https-redirection.config`

```
Resources:
  AWSEBV2LoadBalancerListener:
    Type: AWS::ElasticLoadBalancingV2::Listener
    Properties:
      LoadBalancerArn:
        Ref: AWSEBV2LoadBalancer
      Port: 80
      Protocol: HTTP
      DefaultActions:
        - Type: redirect
          RedirectConfig:
            Host: "#{host}"
            Path: "/#{path}"
            Port: "443"
            Protocol: "HTTPS"
            Query: "#{query}"
            StatusCode: "HTTP_301"
```

`storage-efs-mountfilesystem.config`

```
option_settings:
  aws:elasticbeanstalk:application:environment:
```

```
    FILE_SYSTEM_ID: 'fs-0d392a7309096c187'
    MOUNT_DIRECTORY: '/gitea'

################################################
##### Do not modify values below this line ####
################################################

packages:
  yum:
    amazon-efs-utils: [ ]

commands:
  01_mount:
    command: "/tmp/mount-efs.sh"

files:
  "/tmp/mount-efs.sh":
    mode: "000777"
    content: |
      #!/bin/bash

      EFS_MOUNT_DIR=$(/opt/elasticbeanstalk/bin/get-config environment -k MOUNT_DIRECTORY)
      EFS_FILE_SYSTEM_ID=$(/opt/elasticbeanstalk/bin/get-config environment -k FILE_SYSTEM_ID)

      echo "Mounting EFS filesystem ${EFS_FILE_SYSTEM_ID} to directory ${EFS_MOUNT_DIR} ..."

      echo 'Stopping NFS ID Mapper...'
      service rpcidmapd status &> /dev/null
      if [ $? -ne 0 ] ; then
          echo 'rpc.idmapd is already stopped!'
      else
          service rpcidmapd stop
          if [ $? -ne 0 ] ; then
              echo 'ERROR: Failed to stop NFS ID Mapper!'
              exit 1
          fi
      fi

      echo 'Checking if EFS mount directory exists...'
```

```
if [ ! -d ${EFS_MOUNT_DIR} ]; then
    echo "Creating directory ${EFS_MOUNT_DIR} ..."
    mkdir -p ${EFS_MOUNT_DIR}
    if [ $? -ne 0 ]; then
        echo 'ERROR: Directory creation failed!'
        exit 1
    fi
else
    echo "Directory ${EFS_MOUNT_DIR} already exists!"
fi

mountpoint -q ${EFS_MOUNT_DIR}
if [ $? -ne 0 ]; then
    echo "mount -t efs -o tls ${EFS_FILE_SYSTEM_ID}:/ ${EFS_MOUNT_DIR}"
    mount -t efs -o tls ${EFS_FILE_SYSTEM_ID}:/ ${EFS_MOUNT_DIR}
    if [ $? -ne 0 ] ; then
        echo 'ERROR: Mount command failed!'
        exit 1
    fi
    chmod 777 ${EFS_MOUNT_DIR}
    runuser -l  ec2-user -c "touch ${EFS_MOUNT_DIR}/it_works"
    if [[ $? -ne 0 ]]; then
        echo 'ERROR: Permission Error!'
        exit 1
    else
        runuser -l  ec2-user -c "rm -f ${EFS_MOUNT_DIR}/it_works"
    fi
else
    echo "Directory ${EFS_MOUNT_DIR} is already a valid mountpoint!"
fi

echo 'EFS mount complete.'
```

Configuration Caddy

`CaddyFile`

```
www.drone.nospy.fr {
  redir
```

```
  https:
  //drone.nospy.fr{uri}
}

drone.nospy.fr {
  reverse_proxy
  localhost: 8881
}
```

Boissipay

Dépôt GitHub : https://github.com/Nouuu/boissipay

Context

Le SI d'une agence de voyages est composé de nombreux Microservices.

Éléments fonctionnels

Le responsable du SI souhaite mettre en œuvre un système de paiement différé pour ses clients grands compte (ex : le comité d'entreprise d'une très grande entreprise qui pré-réserve des voyages pour ses collaborateurs).

Le principe est que ces clients, après souscription à un contrat de paiement différé, pourront payer leurs achats de voyages en fin du mois.

Éléments techniques

Ce service de paiement différé sera supporté principalement par deux Microservices : le Microservice de facturation et le Microservice de gestion de contrat.

1. Service de facturation

Le service de facturation accumule des opérations (principalement des achats) et génère à la fin du mois une facture.

Le Microservice de facturation a les fonctions suivantes :

- Acception des opérations (achat ou remboursement)
- Run de facturation : génération d'une facture à la fin du mois pour chacun des clients

Règles de gestion :

- Un achat est accepté selon les conditions suivantes
  - Le contrat est toujours valide

    - Le mandat de prélèvement est toujours valide
    - Le seuil de l’encours n’est pas dépassé

2. Service de contrat

Le service de contrat gère la mastérisation d’un contrat pour son client.

Règles de gestion :

- Diffusion des souscriptions au service de facturation
- Diffusion des changements (suspension/annulation des contrats)

En entrée

- API Swagger Service Facturation
- API Swagger Service de gestion de contrat

Objectif

Décrire et illustrer l’interaction entre ces deux microservices en prenant en compte les problématiques d’usage

Recommandations

1. Ne pas utiliser tous les endpoints REST existant
2. Votre design doit limiter les interactions (ex: le système de facturation doit contenir tous les éléments pour savoir si le contrat est actif)
3. Ne pas oublier de publier

Contraintes

1. Conserver une approche contract-first pour les API REST
    - Possibilité de rajouter /modifier les API données en entrée
2. Utiliser Apache Kafka
    - Un seul broker est suffisant
    - Une seule partition pour chacun des topics est suffisant

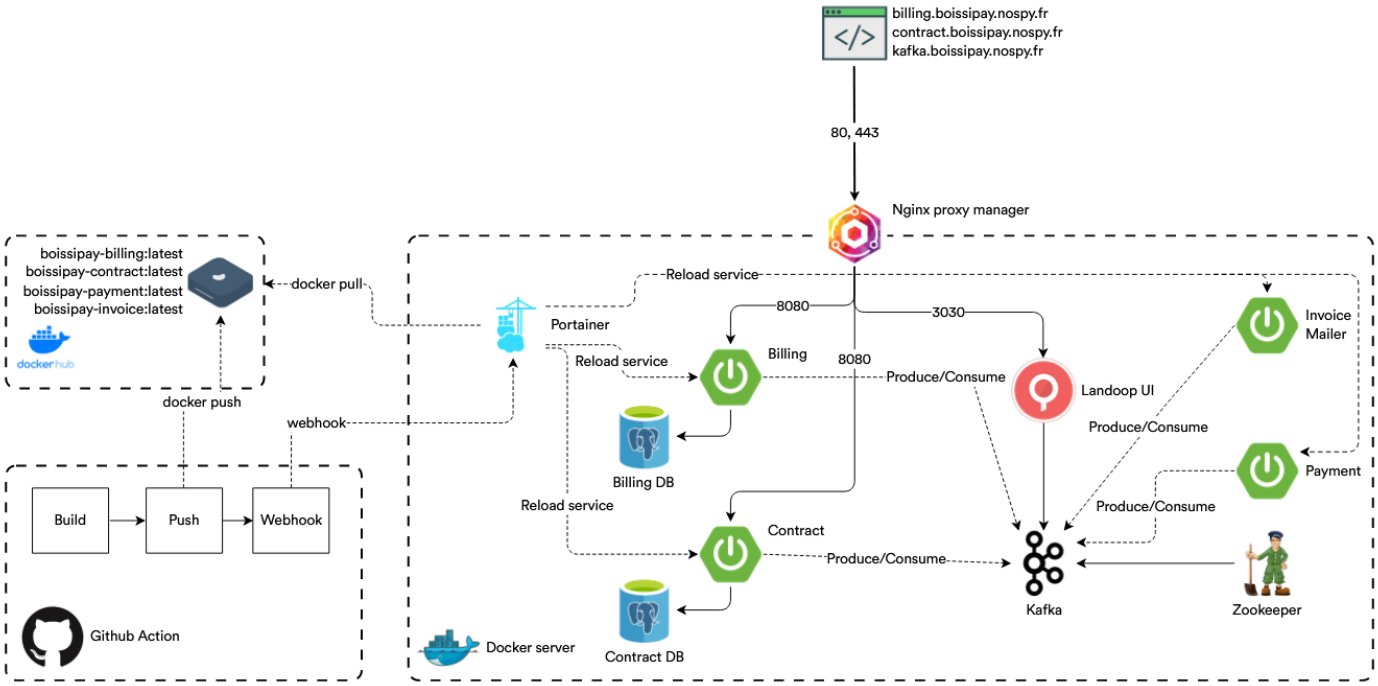
Livrables

- Présentation :
  - Spécifier en détail les usages API ou Event
  - Identifier les choix CQRS
- Code/Démo
  - Se restreindre sur les contrats d'interface, les mécanismes d'interaction
    - Le développement de la logique métier n'est pas attendu
    - Pas de base de données est attendu

Spécifications

Architecture du SI

Voici le schéma de l'architecture du SI de Boissipay :



CI/CD

La CI/CD est géré par :

- **GitHub Action** : Construit l'image docker de tous nos composants, l'envoie sur le dépôt, puis envoie un signal au serveur hébergeant l'application.
- **Docker Hub** : Contient les images docker de tous nos composants.
- **Portainer** : Permet de visualiser l'état des conteneurs. C'est également lui qui contient le manifest du Docker Compose permettant à l'application de se déployer. C'est également lui qui reçoit le signal provenant de GitHub Action afin de mettre à jour les services.

Services

Boissipay est composé de 5 services principaux :

- **Service de facturation** : Permet de gérer les achats et les remboursements.
- **Service de contrat** : Permet de gérer les contrats.
- **Service de paiement** : Permet de gérer les paiements.
- **Service d'envoi de mail** : Permet d'envoyer des mails de facturation aux clients.
- **Kafka** : Permet de gérer les événements entre les différents services.

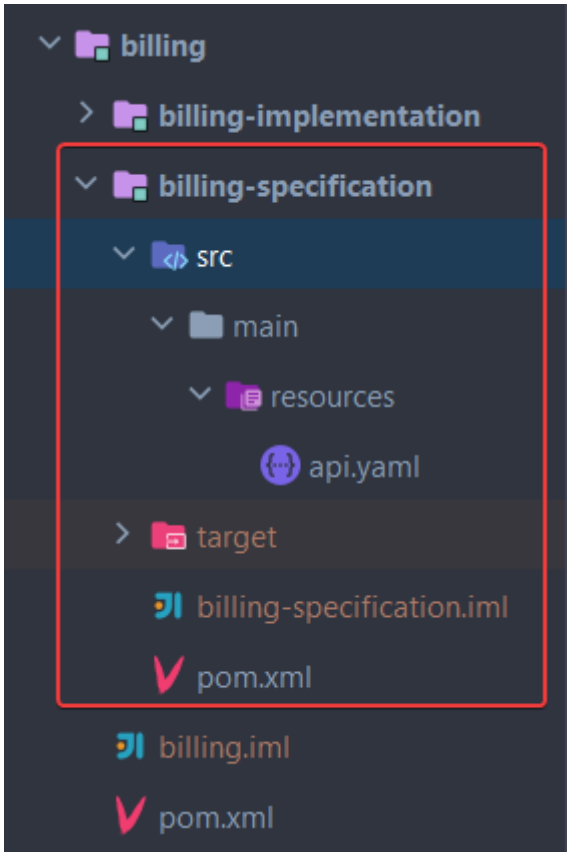
D'autres services sont utilisés pour avoir une application robuste :

- **Base de données des paiements** : Permet de stocker les paiements et leur status.
- **Base de données des contrats** : Permet de stocker les contrats et leur status.
- **Landoop UI** : Permet de visualiser l'état du service Kafka
- **Zookeeper** : Apporte de la persistence au niveau de Kafka.
- **Nginx Proxy Manager** : Permet de gérer le routage vers nos services.

Génération de code (API Delegate)

Afin de rester conforme à la spécification, nous avons mis en place une génération automatique de code pour les API REST. Cela se fait grâce à un module java dédié `Specification` aussi bien pour **Contract** que **Billing** et va générer la partie API de notre application Spring.

Dossier	Pom.xml




```
<build>
    <sourceDirectory>src/main/java</sourceDirectory>
    <plugins>   Pooetitu, 13/06/2022 12:41 · Initial commit
        <plugin>
            <groupId>org.openapitools</groupId>
            <artifactId>openapi-generator-maven-plugin</artifactId>
            <!-- RELEASE_VERSION -->
            <version>6.0.0</version>
            <!-- /RELEASE_VERSION -->
            <executions>
                <execution>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                    <configuration>
                        <apiPackage>org.esgi.boissipay.billing.api</apiPackage>
                        <modelPackage>org.esgi.boissipay.billing.model</modelPackage>
                        <inputSpec>${project.basedir}/src/main/resources/api.yaml</inputSpec>
                        <supportingFilesToGenerate>ApiUtil.java</supportingFilesToGenerate>

                        <generatorName>spring</generatorName>
                        <configOptions>
                            <delegatePattern>true</delegatePattern>
                        </configOptions>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Tout le code métier sera dans le module `Implementation` et viendra surcharger l'API de base généré depuis le Swagger.

```
@Component
public class SpringOperationsApiDelegate implements OperationsApiDelegate {

    // ...

    @Override
    public ResponseEntity<OperationResponse> operationsPost(OperationRequest operationRequest)
{
        String operationId =
newOperationUseCase.newOperation(OperationMapper.toOperation(operationRequest));
        var operation = getOperationUseCase.getOperation(operationId);
        return ResponseEntity.ok(OperationMapper.toOperationResponse(operation));
    }

    // ...
}
```

Docker

Nous avons en place des `Dockerfile` pour chaque composant. Ces derniers prendront en charge certaines variables d'environnement afin de configurer l'application Spring au lancement (Ex : Les identifiants de base de données).

```
FROM maven:3.8.5-eclipse-temurin-17-alpine as build
WORKDIR /app
COPY pom.xml .
COPY . .

RUN mvn clean install -DskipTests

FROM openjdk:17-alpine
WORKDIR /app
COPY --from=build /app/contract/contract-implementation/target/*.jar /app/boissipay-
contract.jar

EXPOSE 8080

ENV KAFKA_HOST=localhost\
    KAFKA_PORT=9092\
    KAFKA_GROUP_ID=boissipay\
    KAFKA_AUTO_OFFSET_RESET=earliest\
    KAFKA_TOPIC_CREATE_CONTRACT=create-contract\
    DATASOURCE_CONNECTION_TIMEOUT=20000\
    DATASOURCE_MAXIMUM_POOL_SIZE=5\
    H2_CONSOLE_ENABLED=true\
    JPA_OPEN_IN_VIEW=false\
    JPA_GENERATE_DDL=true\
    DATABASE_HOST=localhost\
    DATABASE_PORT=5432\
    DATABASE_NAME=contract\
    DATABASE_USER=contract\
    DATABASE_PASSWORD=contract\
    JPA_HIBERNATE_DDL_AUTO=update

CMD ["java", "-jar","-Dspring.profiles.active=env", "/app/boissipay-contract.jar"]
```

Un fichier docker-compose.yml est utilisé pour déployer l'application dans son entièreté sans se prendre la tête.

Des tests de vie et une ordnance de lancement est mis en place pour que chaque service ne démarre que si les prérequis sont remplis.

```
version: '3.8'

services:
  kafka:
    image: landoop/fast-data-dev:latest
    environment:
      ...
    restart: unless-stopped
    networks:
      - boissipay-network
    ports:
      # - 2181:2181               # Zookeeper
      - '3030:3030'               # Landoop UI
      # - 8081-8083:8081-8083     # REST Proxy, Schema Registry, Kafka Connect ports
      # - 9581-9585:9581-9585     # JMX Ports
      # - '9092:9092'
    volumes:
      - 'kafka_data:/data'
    healthcheck:
      test: nc -z localhost 2181 || exit -1
      interval: 10s
      timeout: 10s
      retries: 10

  contract_db:
    image: bitnami/postgresql:14
    environment:
      POSTGRESQL_USERNAME: ${CONTRACT_DB_USERNAME}
      POSTGRESQL_PASSWORD: ${CONTRACT_DB_PASSWORD}
      POSTGRESQL_DATABASE: ${CONTRACT_DB_NAME}
    volumes:
      - contract_db_data:/bitnami/postgresql
    restart: unless-stopped
    networks:
```

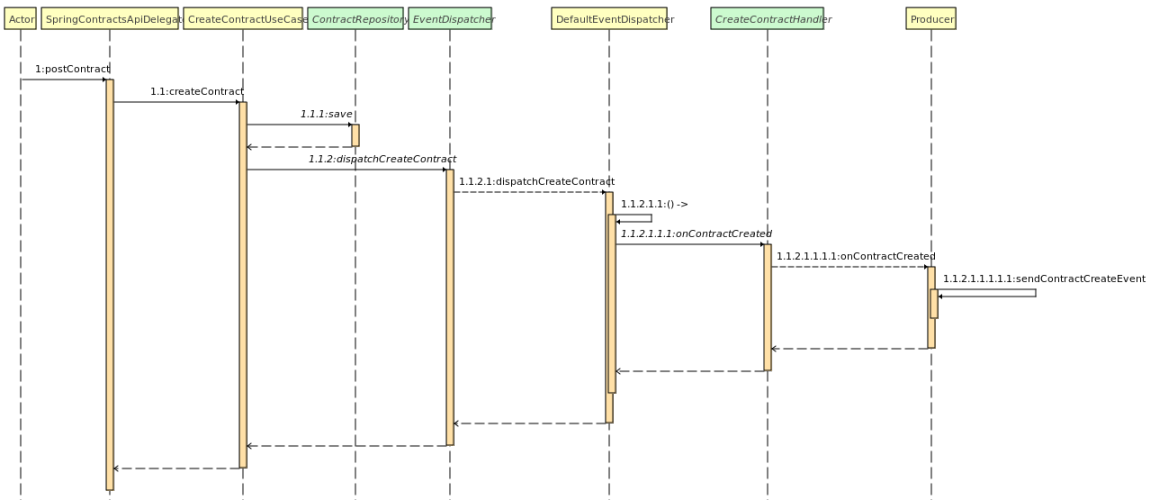
```
    - boissipay-network
  ports:
    - '5433:5432'
  healthcheck:
    test: [ "CMD", "pg_isready", "-U", "contract", "-d", "contract" ]
    interval: 20s
    timeout: 10s
    retries: 5

contract:
  depends_on:
    contract_db:
      condition: service_healthy
    kafka:
      condition: service_healthy
  build:
    context: .
    dockerfile: Dockerfile.contract
  environment:
    KAFKA_HOST: kafka
    KAFKA_PORT: 9092
    KAFKA_GROUP_ID: ${KAFKA_GROUP_ID}
    DATABASE_HOST: contract_db
    DATABASE_PORT: 5432
    DATABASE_NAME: ${CONTRACT_DB_NAME}
    ...
  restart: unless-stopped
  networks:
    - boissipay-network
  ports:
    - '8080:8080'

billing_db:
  ...

billing:
  ...
  restart: unless-stopped
  networks:
```

```
      - boissipay-network
    ports:
      - '8181:8080'

  payment:
    ...

  invoice:
    ...

volumes:
  kafka_data:
  contract_db_data:
  billing_db_data:

networks:
  boissipay-network:
    driver: bridge
```

Contract

Contract est un composant qui va gérer les opérations de création de contrat.

- Lors de la création d'un contrat, ce dernier est enregistré en base de donnée.
- Une fois cela fait, un message est envoyé sur le topic `create-contract` pour que le contrat soit créé sur le Kafka.
- Ce dernier sera consommé plus tard par le module **Billing**.

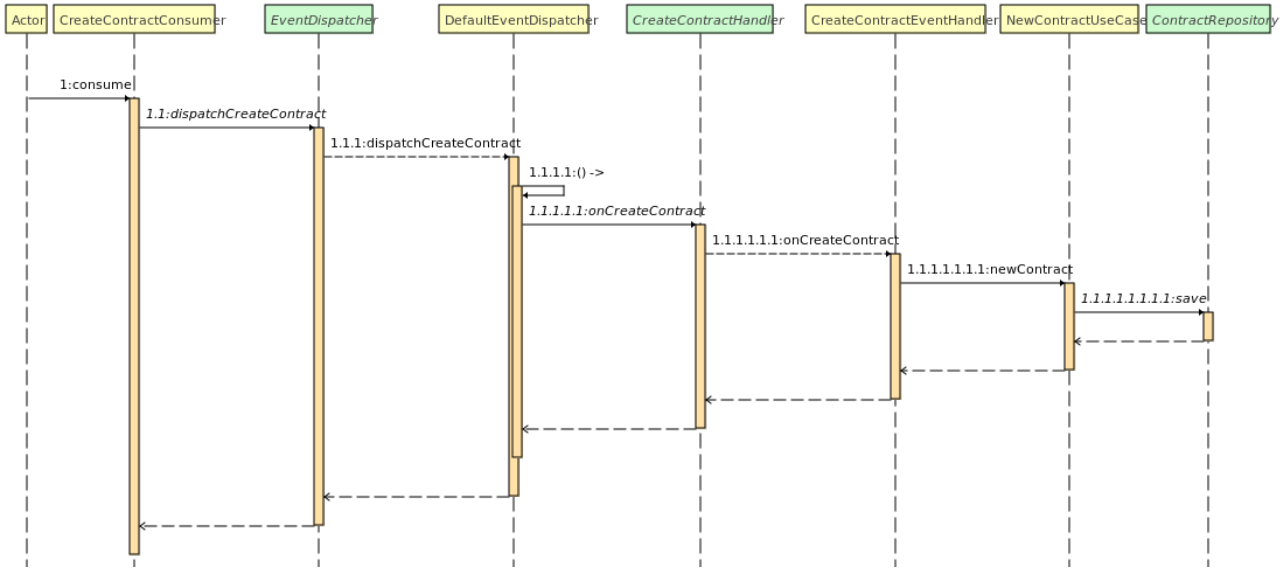


Billing

Billing est un composant qui va gérer les opérations de souscription et de paiements sur un contrat.

Contract Created

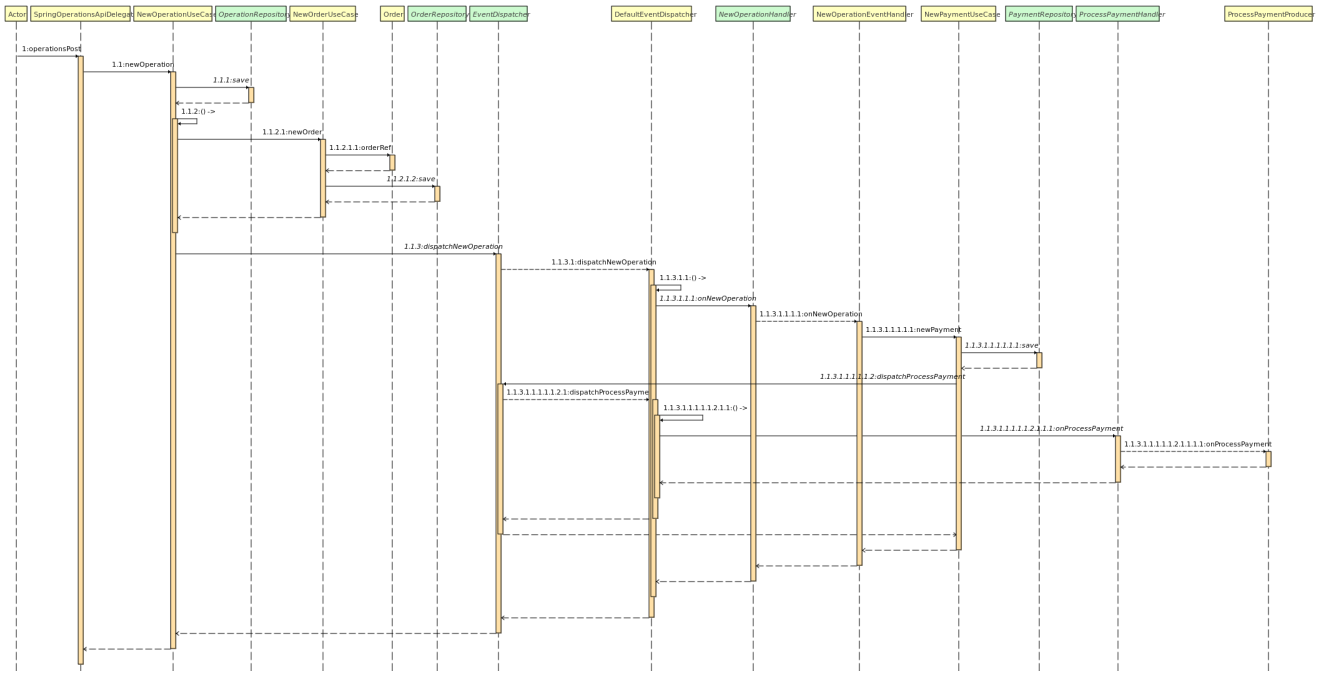
Billing consomme les messages kafka concernant une création de contrat. Lorsqu'il en reçoit un, il va enregistrer en base le contrat créé ainsi que quelques-unes de ses informations nécessaires (Ex: durée de validité).



New operation

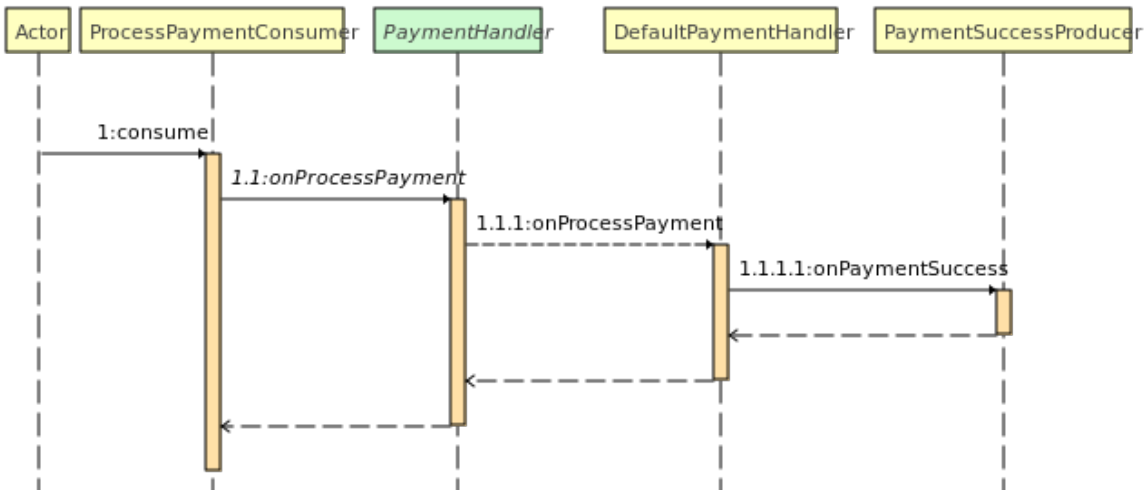
Un utilisateur peut ajouter une nouvelle opération sur un contrat actif. Une opération peut contenir des *Order* qui eux même contiennent des *Items*, ces derniers étant des achats avec un coût.

L'ajout d'une opération va entrainer un enregistrement en base de ce dernier, ainsi que la création d'un paiement (à ce stade non effectué) avec l'envoie d'un message kafka à destination du service de paiement.



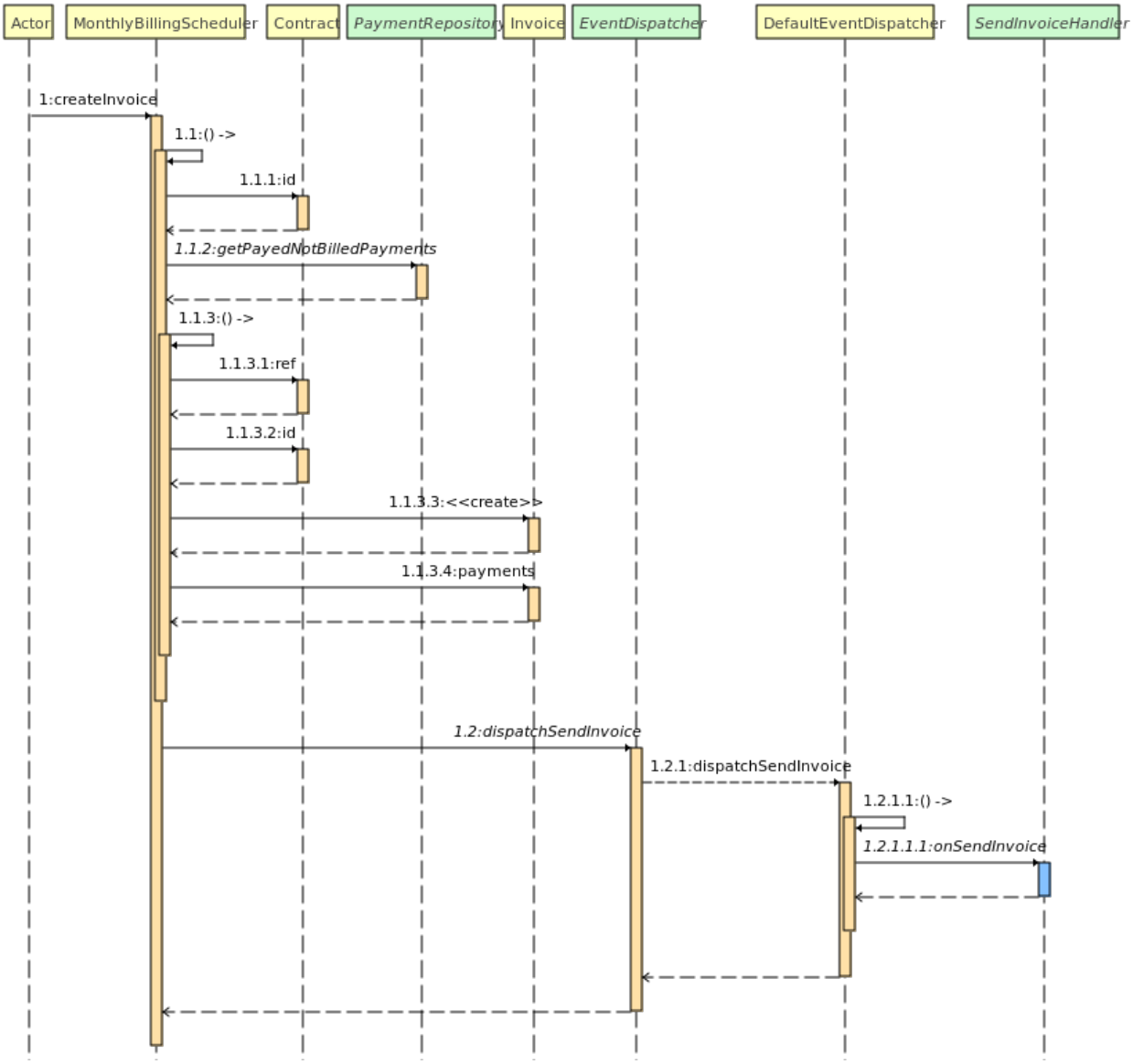
Payment accepted

Lorsqu'un paiement est accepté par le service de paiement, le service **Billing** consomme le message kafka et va mettre à jour le status du paiement afin de le valider.



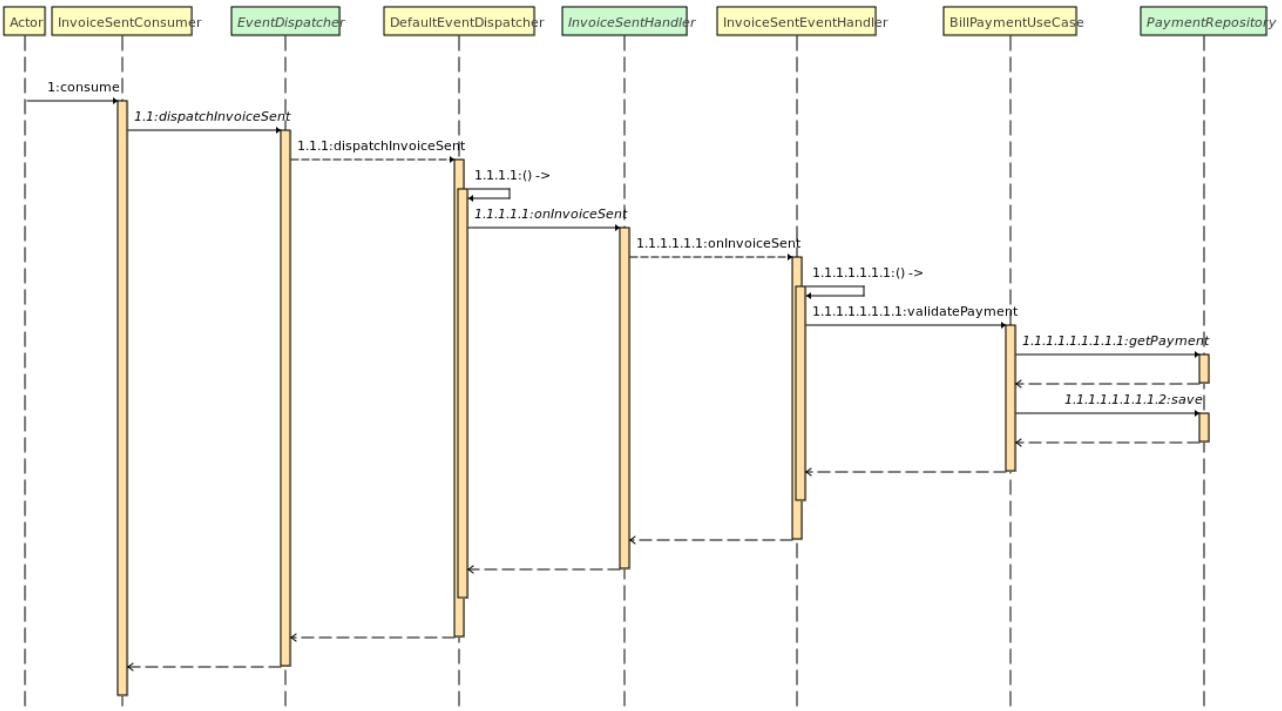
Invoice scheduling

Tous les mois, une facture est envoyée à chaque utilisateur ayant fait des opérations sur un contrat. Le service **Billing** génère la facture en fonction des opérations effectuées sur le contrat. Cette dernière est ensuite envoyée à Kafka pour que le service de facturation puisse la traiter.



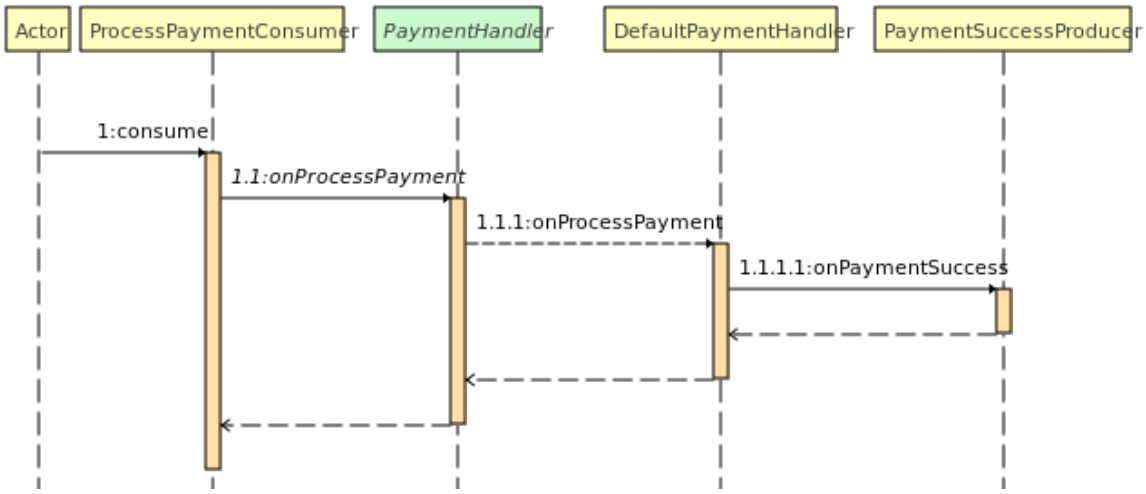
Invoice Sent

Billing possède un `Consumer` Kafka dans le cas où la facture a bien été envoyé au client. Lorsque c'est le cas, on met à jour le status des paiements concerné pour les passer en *Facturé*.



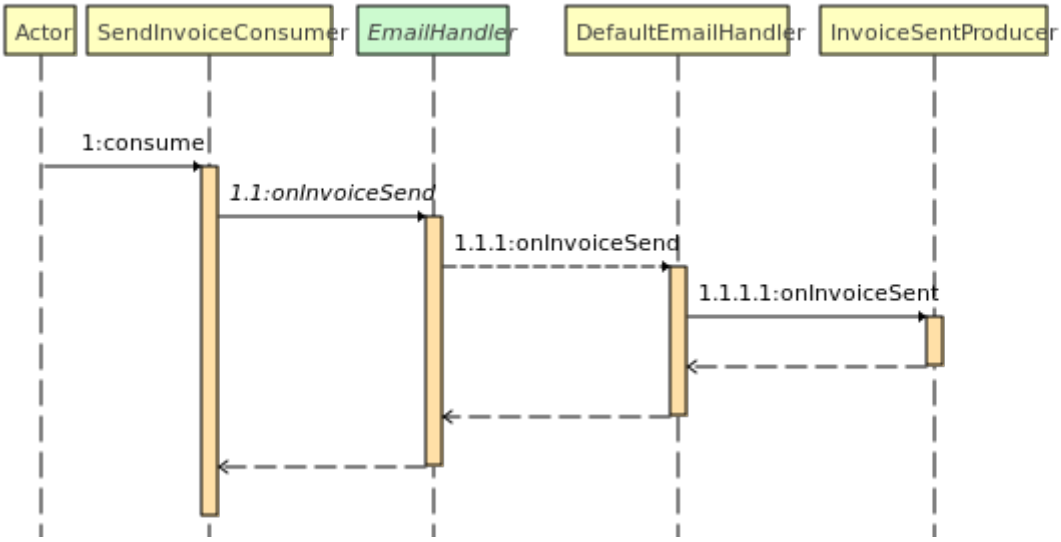
Payment

Payment est un composant qui va gérer les opérations de paiement.
Ce dernier va écouter le topic `payment` et lorsqu'il reçoit un message, il va effectuer le paiement. Le paiement est actuellement simulé et le service va directement envoyé un message Kafka de succès.



Invoice Mailer

Invoice Mailer est un composant qui va gérer l'envoie de facture à l'utilisateur.
Ce dernier va écouter le topic `invoice` et lorsqu'il reçoit un message, il va envoyer la facture à l'utilisateur. Pour le moment aucun mail n'est vraiment envoyé, cela est juste simulé. On renvoie directement un message à Kafka de confirmation.


Actor
SendInvoiceConsumer
EmailHandler
DefaultEmailHandler
InvoiceSentProducer
1:consume
1.1:onInvoiceSend
1.1.1:onInvoiceSend
1.1.1.1:onInvoiceSent