Documentation

Documentation

Lab 2: Prototyping Python

http://bit.ly/ASE23

Lab 2: Prototyping Python

Exam requirements:

Programming skills

- Understand, write, and modify Python code (classes, functions, Flask microservices).
- Use Git from the Command Line Interface to clone, commit, pull and push code.
- Use GitHub to perform a fork and/or a pull request to an existing repository.

Some test examples:

- Given a skeleton repository from GitHub, clone it, extend it with some functionality and push the result on GitHub.
- Fork and perform a pull request to a repo.

Create a repo

- Go to github.com and enter with your credentials: repositories are the place where your projects live.
- In the upper-right corner of any page, click + and then **New Repository**.
- Type a short, memorable name, e.g. ase-fall-22 .
- This repo will be Public.
- Initialise it with a README.
- Click Create a repository.

Clone your repo

- Open GitBash (or bash).
- Move to the directory where you want to store your work (e.g., ASE).
- Use the command `git clone [your repo url] .`
- Create a new folder named Lab_1 and move there the files you want to add to the project folder
- Then:

- `git add *`
- `git commit -m "first commit"`
- `git push`
- Add your team mates as repository collaborators

Manage branches

- Create new branch: `git branch [branch name]`
- Nive to the branch on your local machine: `git checkout [branch name]`
- List all branches: `git branch -a`
- Submit pull requests and merge branches from the web interface by clicking on **Compare & pull request**

Objective: Complete all TODOs in the game.py file without changing the interface. You have to implement 4 functions. Split them across team members and implement one each in 4 different branches of the shared repo. Merge all branches into a single main branch.

Lab 3: Enterprise Integration Pattern with Apache Camel

Exam requirements:

Enterprise Integration Patterns

- Order and compose (given) patterns to implement a business mission via the Apache Camel Java DSL.

Some test examples:

- Given a list of Apache Camel DSL instructions in Java compose them into a program to implement a specific behaviour. The list might contain unnecessary instructions.

See first notes **chapter 4.3** and the deck solution PDF.

Project setup

1. Open IntelliJ IDEA
2. Choose **Maven archetype** on the left

3. Choose a project name
4. Select JDK `openjdk-19`
5. Click on `Add...` to add new archetype for your project
6. Compile the archetype as follows:
    - `GroupId: org.apache.camel.archetypes`
    - `ArtifactId: camel-archetype-java`
    - `Version: 3.19.0`
7. Click on create
8. In IDE's terminal type `mvn install` then launch `mvn camel:run`

Code your route in the `configure()` method of the class `MyRouteBuilder`: **you can use the official documentation**.

```
  public void configure() {
/*Solution--------------------------------------
SPLIT: https://camel.apache.org/components/3.20.x/eips/split-eip.html */
        from("file:" + BASE_PATH + "?noop=true&idempotent=true")
                .split(body().tokenize("\n"))
                .choice().when(simple("${exchangeProperty.CamelSplitIndex} > 0"))// sk
ip headers
                .unmarshal().bindy(BindyType.Csv, SongRecord.class)
                .to("seda:aggregate");

/*
SEDA (camel-seda): Asynchronously call another endpoint from any Camel Context in the
 same JVM.
*/

        from("seda:aggregate?concurrentConsumers=10")
                .bean(MyAggregationStrategy.class, "setArtistHeader")
                .aggregate(new MyAggregationStrategy()).header("artist")
                .completionPredicate(header("CamelSplitComplete").isEqualTo(true))
                .process(new MyProcessor());
/*Solution--------------------------------------*/


    }
```

Lab 4: Docker Compose

Exam requirements:

Docker

- Docker commands to build, tag and run an image.
- Understand, write and modify a `Dockerfile` and issue commands to build and run a Docker image from such file.
- Understand, write and modify a `docker-compose.yml` and issue the commands to build and run a multi-service application with Docker Compose.
- Understand, write and modify a `stack.yml` file and issue the commands to launch a multi- service application in swarm mode.

Some test examples:

- Write a Dockerfile using a specific base image, cloning some code from GitHub, installing dependencies, and launching a Python or bash script. Build and run the image from a Dockerfile.
- Complete a `docker-compose.yml` to launch a simple multi-service application. Build and run the specified orchestration.
- Starting from a `docker-compose.yml`, complete it and convert it into a `stack.yml` file to launch your applications in swarm mode.

Keyword summary

- `FROM python:3.8-slim-buster`: define base image
- `ADD . /gateway`: copy current `dir` into specified one in container
- `WORKDIR /gateway`: move to the new directory
- `RUN pip3 install -r requiremenets.txt`: commands to be run while building image
- `EXPOSE 5000`: container should listen on this port
- `CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]`: specify commands to be launched at startup

Commands summary

- `docker build .`: the dot indicate the path to the Dockerfile
- `docker tag [image ID] [tag name]`: tag an image with a name
- `docker run -p [local-port]:[container-port] [tag name]`

- `docker compose up`: From your project directory, start up your application.
- `docker stack deploy -c stack.yaml name`: to deploy the stack in **Docker Swarm** (after the local machine as been initialized as swarm manager by running `docker swarm init --advertise-addr 127.0.0.1:5001`). Once finished, delete by `docker stack rm name`.

PART ONE

1. Download the code of the microase app from the Moodle.
2. Complete the math service code to feature the requested operations and set a port in the `.flaskenv` file.
3. Run the math service through the provided Dockerfile (instructions up), after completing it with a suitable port.
4. Complete the `gateway/urls.py` file with suitable service name and ports.
5. Write Dockerfile for the `gateway` and `string_rust` services.

PART TWO

1. Write a `docker-compose.yml` file to run all three services.
2. Run the whole application through Docker Compose.

PART THREE (bonus)

1. Add a random service to feature operation related to randomness (e.g. choice, randint, ...).

Dockerfile and `docker-compose.yaml` solutions

Gateway service:

```
FROM python:3.8-slim-buster

ADD . /gateway
WORKDIR /gateway

RUN pip3 install -r requirements.txt

EXPOSE 5000

CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

Math_py service:

```
FROM python:3.8-slim-buster

ADD . /math_py
WORKDIR /math_py

RUN pip3 install -r requirements.txt

EXPOSE 5000

CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

String_rust service:

```
# For the build stage, use the official Rust imageFROM rust:slim-buster AS builder

ARG TARGETPLATFORM

WORKDIR /root

RUN --mount=type=cache,target=/usr/local/cargo/registry,id=${TARGETPLATFORM} \
    cargo install cargo-strip

COPY . .

RUN --mount=type=cache,target=/usr/local/cargo/registry,id=${TARGETPLATFORM} --mount=t
ype=cache,target=/root/target,id=${TARGETPLATFORM} \
    cargo build --release && \
    cargo strip && \
    mv /root/target/release/string_rust /root


FROM debian:buster-slim as runner
COPY --from=builder /root/string_rust /
ENV ROCKET_ADDRESS=0.0.0.0
ENV ROCKET_PORT=5000

EXPOSE 5000

CMD ["./string_rust"]
```

`docker-compose.yml`

```
version: '3'
services:
  math-service:
    build: ./math_py
    container_name: math-service

  string-service:
    build: ./string_rust
    container_name: string-service

  gateway:
    build: ./gateway
    container_name: gateway
    ports:
      - 5000:5000
    depends_on:
      - math-service
      - string-service
```

`stack.yml` for Docker Swarm

```
version: '3.7'

services:
  math-service:
    image: jacopomassa97/math_py:latest
    deploy:
      replicas: 2

  string-service:
    image: jacopomassa97/string_rust:latest
    deploy:
      replicas: 2

  gateway:
    image: jacopomassa97/gateway:latest
    ports:
      - mode: host
        target: 5000
        published: 5001
    depends_on:
      - math-service
      - string-service
```

5. Lab 5: Kubernetes

See chapter 5.3 of notes.

Exam requirements:

Kubernetes

- Issue the commands to start a minikube cluster with K nodes.
- Understand, write and modify deployment.yaml file and a service.yaml files.
- Understand the difference between different type of Object in K8S.

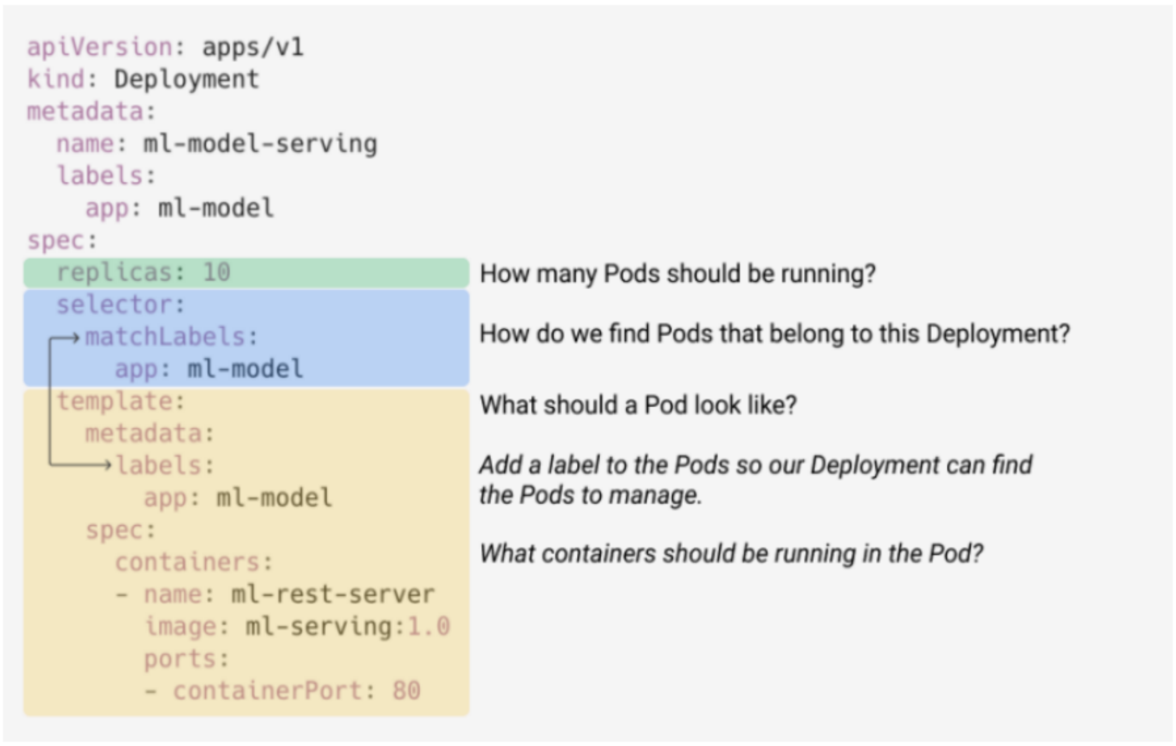
Some test examples:

- None. But there can be questions in the oral part

Command summary

- `minikube start --nodes 2 -p name`: Start a cluster with 2 nodes in the driver of your choice. Run also `minikube tunnel`: the tunnel command exposes the external IP directly to any program running on the host operating system. **Must be executed in a separate terminal.**
- `kubectl get nodes`: get list of your nodes
- `kubectl status -p name`: check status of your node
- Launch the orchestration through commands:
  - `kubectl apply -f deployment.yaml`
  - `kubectl apply -f service.yaml`
- To stop the orchestration issue:
  - `minikube delete --all`
  - `minikube stop --all`

`deployments.yml`



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-model-serving
  labels:
    app: ml-model
spec:
  replicas: 10
  selector:
    matchLabels:
      app: ml-model
  template:
    metadata:
      labels:
        app: ml-model
    spec:
      containers:
      - name: ml-rest-server
        image: ml-serving:1.0
        ports:
        - containerPort: 80
```

How many Pods should be running?

How do we find Pods that belong to this Deployment?

What should a Pod look like?

Add a label to the Pods so our Deployment can find the Pods to manage.

What containers should be running in the Pod?

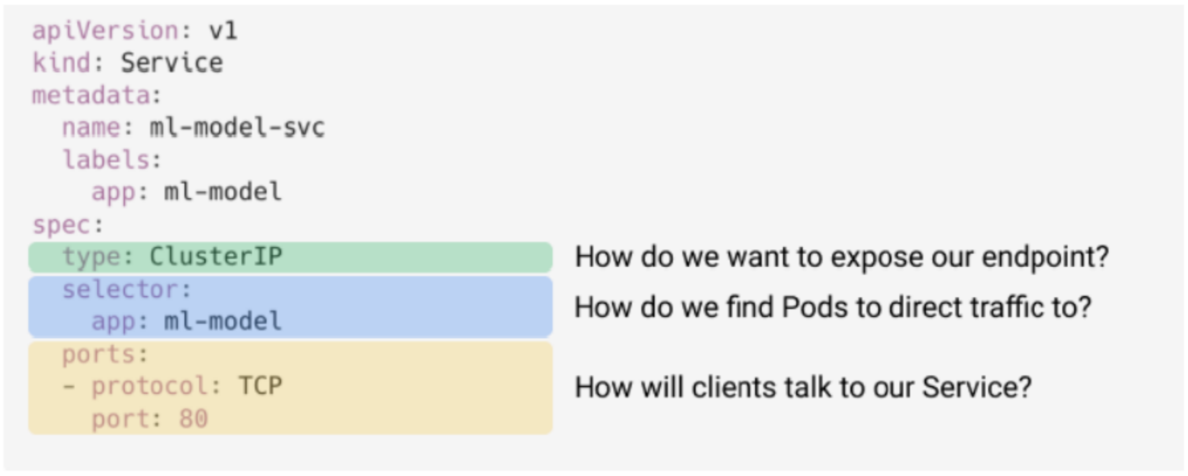
Figure 5.11: `deployment.yaml`

The `.metadata` section is where we give the Deployment a name and labels. The .`spec` section is where most of the action happens. Anything directly below .`spec` relates to the Pod. Anything nested below `.spec.template` relates to the Pod template that the Deployment will manage. In this example, the Pod template defines a single container.

- `.spec.replicas` tells Kubernetes how may Pod replicas to deploy.
- `spec.selector` is a list of labels that Pods must have in order for the Deployment to manage them.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gateway
  template:
    metadata:
      labels:
        app: gateway
    spec:
      containers:
      - name: gateway
        image: jacopomassa97/gateway:latest
        ports:
          - containerPort: 5000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: math-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: math-service
  template:
    metadata:
      labels:
        app: math-service
    spec:
      containers:
      - name: math-service
        image: jacopomassa97/math_py:latest
        ports:
          - containerPort: 5000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: string-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: string-service
  template:
    metadata:
      labels:
        app: string-service
    spec:
      containers:
      - name: string-service
        image: jacopomassa97/string_rust:latest
        ports:
          - containerPort: 5000
```

`service.yml`



```yaml
apiVersion: v1
kind: Service
metadata:
  name: ml-model-svc
  labels:
    app: ml-model
spec:
  type: ClusterIP
  selector:
    app: ml-model
  ports:
  - protocol: TCP
    port: 80
```

How do we want to expose our endpoint?

How do we find Pods to direct traffic to?

How will clients talk to our Service?

Figure 5.13: `service.yaml`

Kubernetes `ServiceTypes` allow you to specify what kind of Service you want. Type values and their behaviors are:

- `ClusterIP`: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a type for a Service.
- `NodePort`: Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: ClusterIP.
- `ExternalName`: Exposes the Service externally using a cloud provider's load balancer.
- `LoadBalancer`: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: gateway
spec:
  type: LoadBalancer
  selector:
    app: gateway
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: math-service
spec:
  selector:
    app: math-service
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: string-service
spec:
  selector:
    app: string-service
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
```

Lab 6: Refactoring and smells with MicroFreshener

Exam requirements:

Smells and refactorings

- Launch and use MicroFreshener to identify and fix smells in microservice-based architectures.

Some test examples:

- Analyse a given application and apply refactorings suggested from MicroFreshener. Simplify, if possible, the suggested refactorings.

Quick Guide

In order to run the `MicroFreshener` you should run the server and the client (for local use only).

Installation

In order to use `MicroFreshener` you should first download the repository:

```
git clone https://github.com/di-unipi-socc/microFreshener.git
```

Run the server

Enter in the server directotry

```
cd /server
```

Create a virtual environment and install the python dependencies

```
virtualenv -p python3 venv
source venv/bin/activate
pip install -r requirements.txt
```

Apply migrations

```
python manage.py migrate
```

Run the local server

```
python manage.py runserver

// expose the server to a specific port
python manage.py runserver 0.0.0.0:8000
```

Run the client

The client is an Angular web application.

```
cd /client
```

Serve the client with a local server

```
sudo ng serve
```

Open the bowser on http://127.0.0.1:4200/.

with docker compose

Make sure to install the following dependencies:

```
pip3 install websocket
pip3 install docopt
pip3 install texttable
pip3 install dockerpty
pip3 install websocket-client==0.32.0
```

Create the docker network

```
docker network create web
```

Build the image

```
docker-compose -f docker-compose.prod.yml build
```

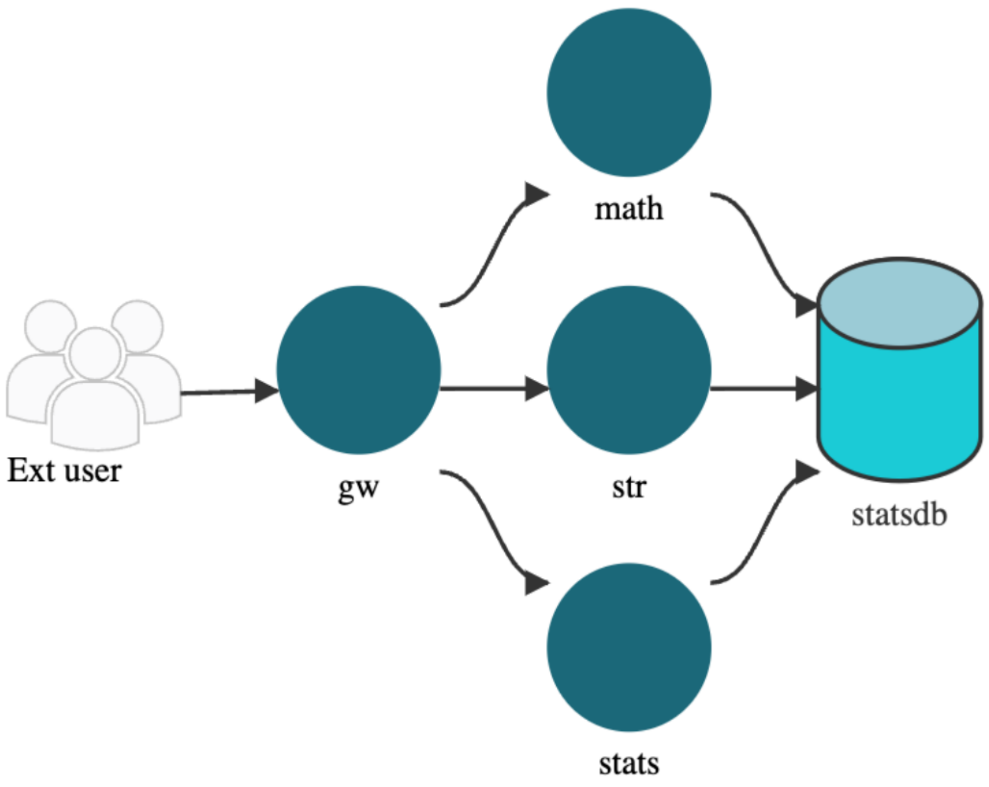
```
docker-compose -f docker-compose.prod.yml up
```

Open the bowser on http://127.0.0.1.

Go to `File -> Examples -> Hello world`

Project steps

1. Download `microase` from moodle
2. Use MicroFreshener to draw the architecture of microase in microTosca

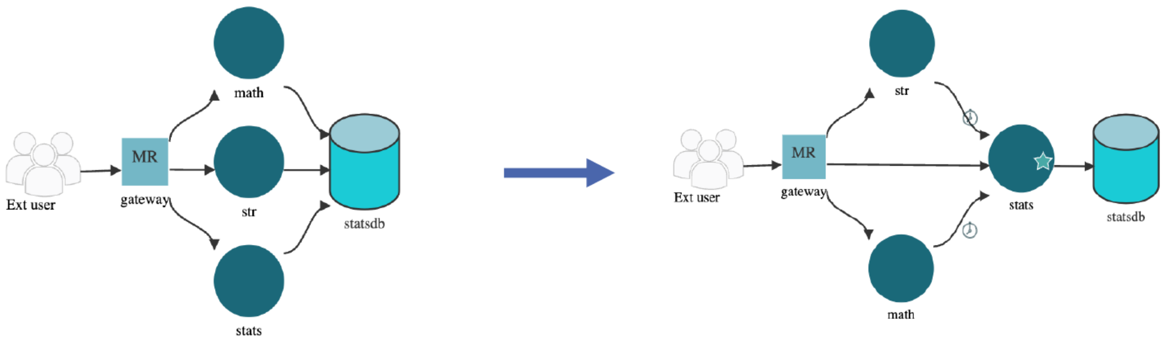


3. Use MicroFreshener to identify smells and possible refactorings:
    - `Actions -> Analyse`
    - Save

Teamwork goals:

The only smell that needs to be addressed by implementing a different architecture is the Shared Persistency among stats, math and str.

EXERCISE : Write code to fix such smell by implementing the architectural refactoring suggested by Freshener.



Lab7: Static analysis with Bandit

Exam requirements:

Security analyses

- Launch static security analysis with bandit and understand the output.

Some test examples:

- Analyse a given application with bandit and fix a simple security issue (e.g. hardcoded password).

Steps

1. `pip install bandit`
2. `bandit -r .`
3. Download `microase` and dice-with-rolls.
4. Scan it with Bandit.
5. Discover and fix vulnerabilities. On `dice-with-rolls` return:

```
[main]  INFO  profile include tests: None
[main]  INFO  profile exclude tests: None
[main]  INFO  cli include tests: None
[main]  INFO  cli exclude tests: None
[main]  INFO  running on Python 3.8.10
64 [0.. 50.. ]
Run started:2023-01-05 14:19:15.512887

Test results:
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'A SECRET KEY'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-api-gateway/APIGateway/app.py:11:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
10      flask_app.config['TESTING'] = True
11      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
12      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'ANOTHER ONE'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-api-gateway/APIGateway/app.py:12:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
11      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
```

```
12      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'
13      flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: ''
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-api-gateway/APIGateway/classes/User.py:13:15
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
12      email = ""
13      password = ""
14
15      def __init__(self, id, firstname, lastname, email):

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'A SECRET KEY'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-dice/dice_service/app.py:12:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
11      flask_app.config['TESTING'] = (database == TEST_DB)
12      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
13      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'ANOTHER ONE'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-dice/dice_service/app.py:13:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
12      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
13      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'
14      flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

--------------------------------------------------
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for secu
rity/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   Location: ./dwr-dice/dice_service/database.py:49:29
   More Info: https://bandit.readthedocs.io/en/1.7.4/blacklists/blacklist_calls.html#b
311-random
48      def roll_die(self):
49          return {self.number: rnd.choice(self.figures_to_array())}
50

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'A SECRET KEY'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-reactions/ReactionsService/app.py:11:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
10      flask_app.config['TESTING'] = True
11      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
```

```
12      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'ANOTHER ONE'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-reactions/ReactionsService/app.py:12:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
11      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
12      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'
13      flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

--------------------------------------------------
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through strin
g-based query construction.
   Severity: Medium   Confidence: Low
   CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
   Location: ./dwr-reactions/ReactionsService/views/reactions.py:112:12
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b608_hardcoded_sql_expres
sions.html
111         "SELECT reaction_caption FROM reaction_catalogue ORDER BY reaction_captio
n").fetchall()
112     query = "SELECT reaction_caption, counter FROM counter c, reaction_catalogue r
WHERE " \
113             "reaction_type_id = reaction_id AND story_id = " + str(story_id) + " O
RDER BY reaction_caption "
114     story_reactions = db.engine.execute(query).fetchall()

--------------------------------------------------
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through strin
g-based query construction.
   Severity: Medium   Confidence: Medium
   CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
   Location: ./dwr-reactions/ReactionsService/views/reactions.py:149:39
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b608_hardcoded_sql_expres
sions.html
148         for story in all_stories:
149             result = db.engine.execute("SELECT sum(counter) as num_reactions "
150                                        "FROM counter "

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'A SECRET KEY'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-stories/StoriesService/app.py:13:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
12      flask_app.config['TESTING'] = True
13      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
14      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'ANOTHER ONE'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-stories/StoriesService/app.py:14:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
```

```
tring.html
13      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
14      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'
15      flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

--------------------------------------------------
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for secu
rity/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   Location: ./dwr-stories/StoriesService/views/stories.py:217:14
   More Info: https://bandit.readthedocs.io/en/1.7.4/blacklists/blacklist_calls.html#b
311-random
216     else:
217         pos = randint(0, len(recent_stories) - 1)
218         return jsonify(recent_stories[pos].to_json())

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'A SECRET KEY'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-users/UsersService/app.py:12:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
11      flask_app.config['TESTING'] = True
12      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
13      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'

--------------------------------------------------
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'ANOTHER ONE'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   Location: ./dwr-users/UsersService/app.py:13:21
   More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b105_hardcoded_password_s
tring.html
12      flask_app.config['WTF_CSRF_SECRET_KEY'] = 'A SECRET KEY'
13      flask_app.config['SECRET_KEY'] = 'ANOTHER ONE'
14      flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

--------------------------------------------------

Code scanned:
  Total lines of code: 3811
  Total lines skipped (#nosec): 0

Run metrics:
  Total issues (by severity):
    Undefined: 0
    Low: 13
    Medium: 2
    High: 0
  Total issues (by confidence):
    Undefined: 0
    Low: 1
    Medium: 12
    High: 2
Files skipped (0):
```

Lab 8: Dynamic analysis with OSWAP

Exam requirements:

Security analyses

- Launch penetration with OWASP Zap

Some test examples:

- Launch a microservice-based application with Docker Compose and analyse it with OWASP Zap in automated/manual mode.

Steps

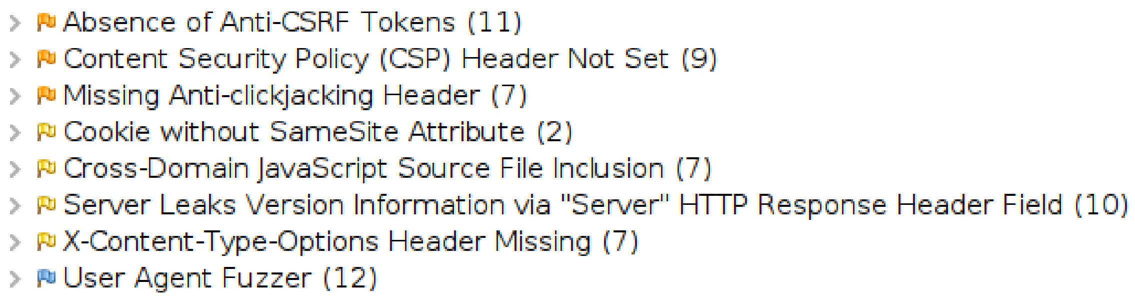
1. Run ZAP.
2. Select the configuration and click 'Start'.

Automated scan

- Disconnect from internet

1. Provide the URL (http://localhost:9090/SERVICE) as the url to attack
2. Click attack

▼ [AUTOMATED] For dice-with-rolls microservices



Absence of Anti-CSRF Tokens (11)
Content Security Policy (CSP) Header Not Set (9)
Missing Anti-clickjacking Header (7)
Cookie without SameSite Attribute (2)
Cross-Domain JavaScript Source File Inclusion (7)
Server Leaks Version Information via "Server" HTTP Response Header Field (10)
X-Content-Type-Options Header Missing (7)
User Agent Fuzzer (12)

▼ [MANUAL] For dice-with-rolls microservices

Manual scan

- Disconnect from internet

1. Restart ZAP
2. Click on Manual Explore
3. Same URL as before, check `Enable HUD` and click launch browser (as browser `Firefox`).
4. (**FOR WebGoat**): Login with the account you have previously created
5. (**FOR WebGoat**): Click the «Cross-site Scripting» link and go at step 5
6. You get notified about a possible XSS vulnerability.

Vulnerabilities are also described in the UI. To perform an XSS attack on **WebGoat** Write some code in the field that accepts credit card numbers like `<script>alert("Attacked");</script>`

Lab 9: CICD with Jenkins

Exam requirements:

Jenkins

- Use Jenkins to automate simple CI/CD pipelines (build, test, deliver).
- Understand, write and modify a simple Jenkinsfile.

Some test examples:

- Complete/modify a given Jenkins file so to perform CI/CD on a given code repository.

▼ Installation & execution steps

1. Create a bridge network for Jenkins to communicate with containers via `docker network create jenkins`
2. Run docker in docker `dind`:

```
docker run \
--name jenkins-docker \
--rm \
--detach \
--privileged \
--network jenkins \
--network-alias docker \
--env DOCKER_TLS_CERTDIR=/certs \
--volume jenkins-docker-certs:/certs/client \
--volume jenkins-data:/var/jenkins_home \
docker:dind \
--storage-driver overlay2
```

1. Write Jenkins official `Dockerfile`:

```
FROM jenkins/jenkins:2.361.4-jdk11
USER root
RUN apt-get update && apt-get install -y lsb-release
RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \
https://download.docker.com/linux/debian/gpg
RUN echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \
https://download.docker.com/linux/debian \
$(lsb_release -cs) stable" > /etc/apt/sources.list.d/docker.list
RUN apt-get update && apt-get install -y docker-ce-cli
USER jenkins
RUN jenkins-plugin-cli --plugins "blueocean:1.25.8 docker-workflow:521.v1a_a_dd207
3b_2e"
```

1. Build it: `docker build -t myjenkins-blueocean:2.361.4-1 .`
2. Run it:

```
docker run \
--name jenkins-blueocean \
--detach \
--network jenkins \
--env DOCKER_HOST=tcp://docker:2376 \
--env DOCKER_CERT_PATH=/certs/client \
--env DOCKER_TLS_VERIFY=1 \
--publish 8080:8080 \
--publish 50000:50000 \
--volume jenkins-data:/var/jenkins_home \
--volume jenkins-docker-certs:/certs/client:ro \
--volume "$HOME":/home \
--restart=on-failure \
--env JAVA_OPTS="-Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true" \
myjenkins-blueocean:2.361.4-1
```

1. Access jenkins on http://127.0.0.1:8080
2. Retrieve pasword by ussing the command `docker logs jenkins-blueocean` (the password is an hash-like strings showed in the log prompt)

3. Click **Install suggested plugins**
4. Create admin user and continue until **Start using Jenkins**
5. Clone the given project `git clone https://github.com/jenkins-docs/simple-python-pyinstaller-app.git`. The folder where you clone is relevent for Jenkins to recognize the .git repo. Your home directory, e.g. `username/...` will be mapped to `home/...`
6. Create a pipeline:
    - New Element -> Pipeline
    - Choose the definition "Pipeline script from SCM"
    - From the SCM field, choose Git.
    - In the Repository URL field, specify the directory path of your locally cloned repository above, which is from your user account/home directory on your host machine, mapped to the /home directory of the Jenkins container - i.e.
        - For macOS - /home/Documents/ase/simple-python-pyinstaller-app
        - For Linux/WSL - /home/ase/simple-python-pyinstaller-app
        - For Windows - /home/Documents/ase/simple-python-pyinstaller-app
    - Click Save

    Pipeline Syntax

    This video shares some differences between Scripted and Declarative Pipeline syntax. When Jenkins Pipeline was first created, Groovy was selected as the foundation. Jenkins

    https://www.jenkins.io/doc/book/pipeline/syntax/

    Following script are taken from official tutorial at:

    Build a Python app with PyInstaller

    FROM jenkins/jenkins:2.375.2 USER root RUN apt-get update && apt-get install -y lsb-release RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \

    https://www.jenkins.io/doc/tutorials/build-a-python-app-with-pyinstaller/

7. Create a Jenkins file:

```
pipeline {
  agent none
  stages {
    stage('Build') {
      agent {
        docker {
          image 'python:2-alpine'
        }
      }
      steps {
        sh 'python -m py_compile sources/add2vals.py sources/calc.py'
        stash(name: 'compiled-results', includes: 'sources/*.py*')
      }
    }
  }
}
```

Then commit:

```
git add .
git commit –m "Add initial Jenkinsfile"
```

1. Open Blue Ocean with the button Open Blue Ocean In the This job has not been run message box, click Run or “Build Now” inside the pipeline dashboard (outside Blue Ocean)
2. **Add test stage**:

```
stage('Test') {
  agent {
    docker {
      image 'qnib/pytest'
    }
  }
  steps {
    sh 'py.test --junit-xml test-reports/results.xml sources/test_calc.py'
  }
  post {
    always {
      junit 'test-reports/results.xml'
    }
  }
}
```

Then commit

```
git add .
git commit –m "Add ‘Test’ stage"
```

And rerun the job in Jenkins (via the Open Blue Ocean button).

1. **Add final deliver stage**:

```
stage('Deliver') {
  agent any
  environment {
    VOLUME = '$(pwd)/sources:/src'
    IMAGE = 'cdrx/pyinstaller-linux:python2'
  }
  steps {
    dir(path: env.BUILD_ID) {
      unstash(name: 'compiled-results')
      sh "docker run --rm -v ${VOLUME} ${IMAGE} 'pyinstaller -F add2vals.py'"
    }
  }
  post {
    success {
      archiveArtifacts "${env.BUILD_ID}/sources/dist/add2vals"
      sh "docker run --rm -v ${VOLUME} ${IMAGE} 'rm -rf build dist'"
    }
  }
}
```

Then commit:

```
git add .
git commit –m "Add ‘Deliver’ stage"
```

And rerun the job in Jenkins (via the Open Blue Ocean button).

The final `Jenkinsfile` for the app tic-tac-toe is the following:

```
pipeline {
  agent none
  options {
    skipStagesAfterUnstable()
  }
  stages {
    stage('Build') {
      agent {
        docker {
          image 'python:3-alpine'
        }
      }
      steps {
        sh 'python3 -m py_compile game.py board.py'
        stash(name: 'compiled-results', includes: '*.py*')
      }
    }
    stage('Test') {
      agent {
        docker {
          image 'grihabor/pytest:python3.7-alpine'
        }
      }
      steps {
        sh 'py.test --junit-xml test-reports/results.xml test_game.py'
      }
      post {
        always {
          junit 'test-reports/results.xml'
        }
      }
    }
  }
}
```

Exercise

- Clone the repo at https://github.com/teto1992/tic-tac-toe
- Write a Jenkins pipeline to build tic-tac-toe into two stages **Build** and **Test**. Choose a Docker images for building and testing featuring Python2.

Solution:

```
pipeline {
    agent none
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:3-alpine'
                }
            }
            steps {
                sh 'python3 -m py_compile game.py board.py'
                stash(name: 'compiled-results', includes: '*.py*')
            }
        }
        stage('Test') {
            agent {
                docker {
                    image 'grihabor/pytest:python3.7-alpine'
                }
            }
            steps {
                sh 'py.test --junit-xml test-reports/results.xml test_game.py'
            }
            post {
                always {
                    junit 'test-reports/results.xml'
                }
            }
        }
    }
}
```

Lab 10: Camunda Modeler

Exam requirements:

- Use the Camunda Modeler to write/modify BPMNs.
- Deploy and run process instances with the Camunda Engine.
- Understand the two usage patterns of Camunda and code simple workers for pattern B.
- Create/modify a WF-net and perform automated analyses of WF-nets with WoPed.

Some test examples:

- Deploy a BPMN file to Camunda from the Camunda Modeler and complete the code of a worker to accomplish a specific task. Launch a process instance and the worker.
- Transform a BPMN process into a WF-net and analyse it with WoPed.

Steps

1. Use Docker to run Camunda BPM Platform:

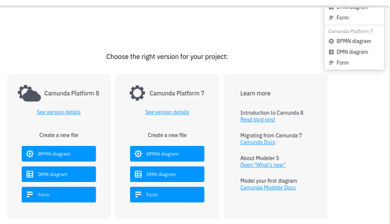
```
docker pull camunda/camunda-bpm-platform:latest
docker run -d --name camunda -p 8080:8080 camunda/camunda-bpm-platform:latest
```

1. Browse `127.0.0.1:8080/camunda` and enter credentials `demo demo`.
2. Install the Camunda Modeler from https://camunda.com/download/modeler/
3. Start the modeler: Click `BPMN Diagram` selecting **Camunda 7**
4. Design the model
   a.

Executing automated steps (2/6)

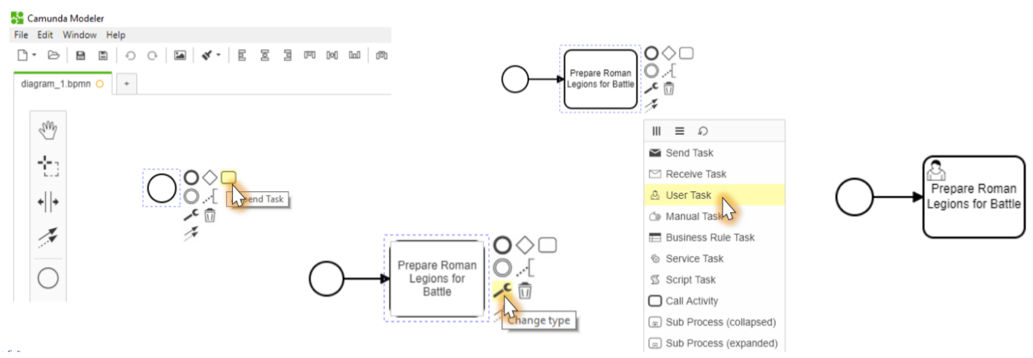
In this section, you'll learn how to create your first BPMN 2.0 process with the Camunda Modeler and how to execute automated steps. Start by opening up Camunda Modeler.

https://docs.camunda.org/get-started/quick-start/service-task/#b-using-javascript-nodejs

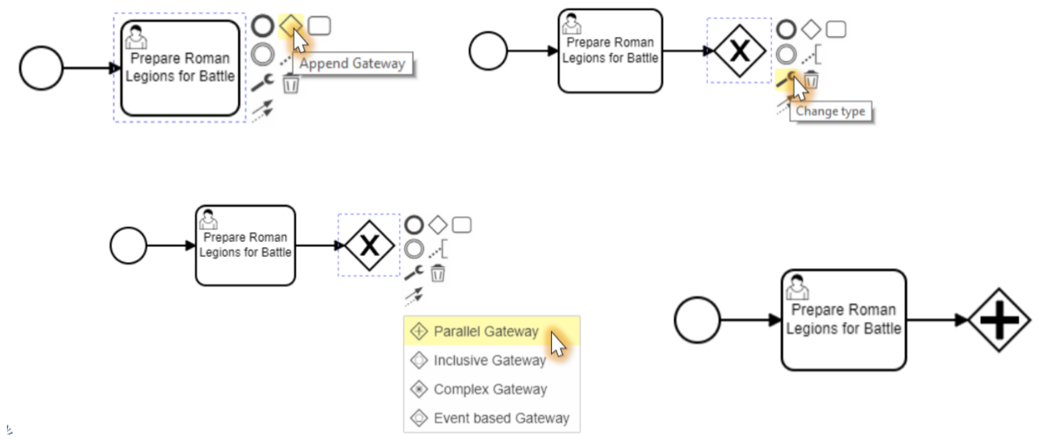


▼ Roman Legions Model

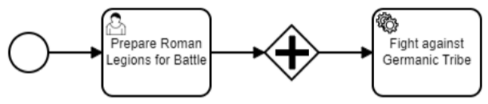
1. **To create a User Task (i.e., involving humans**



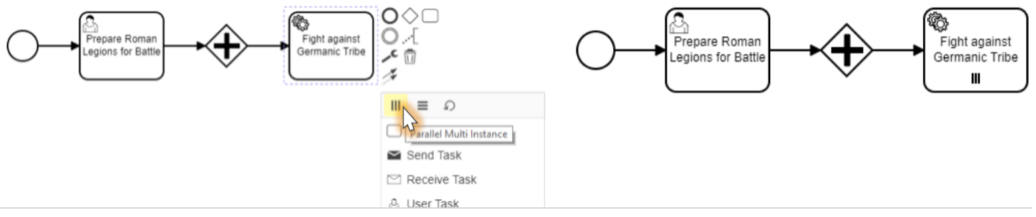
2. **Add a parallel gateway**


Prepare Roman Legions for Battle
Append Gateway
Prepare Roman Legions for Battle
Change type
Prepare Roman Legions for Battle
Parallel Gateway
Inclusive Gateway
Complex Gateway
Event based Gateway
Prepare Roman Legions for Battle

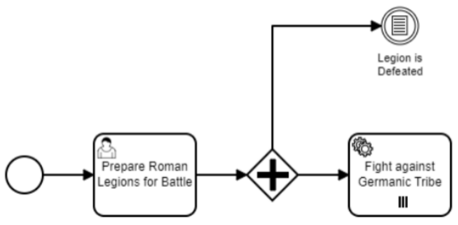
3. **Add a Service Task:** Simply select the **Service Task** type


Prepare Roman Legions for Battle
Fight against Germanic Tribe

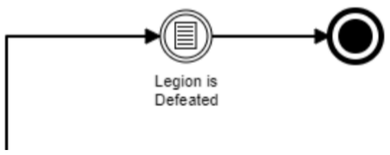
- Then, make it **Parallel Multi-Instance** (we can have many Germanic Tribes attacking our Legion).


Prepare Roman Legions for Battle
Fight against Germanic Tribe
Parallel Multi Instance
Send Task
Receive Task
User Task
Prepare Roman Legions for Battle
Fight against Germanic Tribe

Add an **Intermediate Event** and make it **Conditional** (what if we lose the battle?)


Legion is Defeated
Prepare Roman Legions for Battle
Fight against Germanic Tribe

Append an **End Event** and make it a **Terminate Event**.



4.

5. Save and click on `Deploy Current Diagram`

6. Enter in Camunda Cockpit and click on Processes at the top of the screen. **External task** exploit the REST API:

    - Fetch and Lock: `POST /external-task/fetchAndLock`, Ask for a task to compute (and lock it to prevent other worker from doing the job)
    - Complete: `POST /external-task/{id}/complete`, Tell that a task has been completed to Camunda

- To submit a Task visit **Camunda Tasklist** (`http://127.0.0.1:PORT/camunda/app/tasklist`) and click on Start Process on top right and Select the Process (e.g. RomanLegion). This will create task to be execute from external worker.

1. Create Javascript workers:

```
mkdir romanlegions
cd ./romanlegions
npm init -y
npm install -s camunda-external-task-client-js
```

The `worker.js` code is:

```
const {
  Client,
  logger,
  Variables
} = require("camunda-external-task-client-js");

// configuration for the Client://  - 'baseUrl': url to the Workflow Engine//  - 'logg
er': utility to automatically log important eventsconst config = {
  baseUrl: "http://localhost:8080/engine-rest",
  use: logger
};

// create a Client instance with custom configurationconst client = new Client(confi
g);

const handler = async({ task, taskService }) => {
// Put your business logic here

};

// susbscribe to the topic: "FightTribe"
client.subscribe("FightTribe", handler);
```

The solution is:

```
const {
    Client,
    logger,
    Variables
  } = require("camunda-external-task-client-js");

// configuration for the Client://  - 'baseUrl': url to the Workflow Engine//  - 'logg
er': utility to automatically log important eventsconst config = {
    baseUrl: "http://127.0.0.1:8080/engine-rest",
    use: logger
  };

// create a Client instance with custom configurationconst client = new Client(confi
g);

  const handler = async({ task, taskService }) => {

    const variables = new Variables()
    if (Math.random() > 0.5){
        console.log("[Germanic Tribe Fighter] The battle has lost!");
        Variables.set("legionStatus", "defeated");
    } else {
        console.log("[Germanic Tribe Fighter] The Roman legion has won the battle!");
        Variables.set("legionStatus", "victorious");
    }

//Complete the tasktry{
        await taskService.complete(task, variables);
        console.log("Task completed.")

    } catch(e){
        console.error(`Failed completing the task ${e}`)
    }

  };

// susbscribe to the topic: "FightTribe"
  client.subscribe("FightTribe", handler);
```

Lab 11: WoPeD

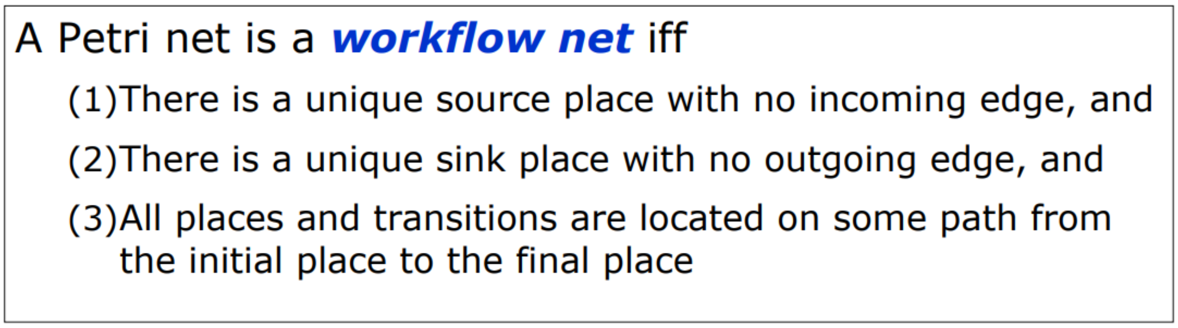
Exam requirements:

- Create/modify a WF-net and perform automated analyses of WF-nets with WoPed.

Some test examples:

- Transform a BPMN process into a WF-net and analyse it with WoPed.

Definitions

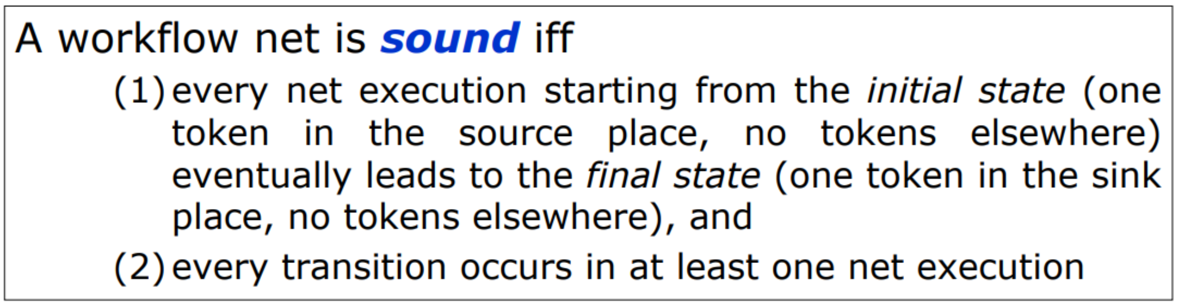


A Petri net is a ***workflow net*** iff

(1)There is a unique source place with no incoming edge, and

(2)There is a unique sink place with no outgoing edge, and

(3)All places and transitions are located on some path from the initial place to the final place



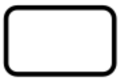
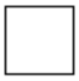
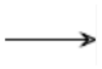
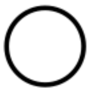
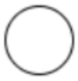
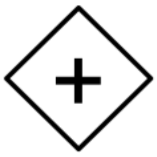
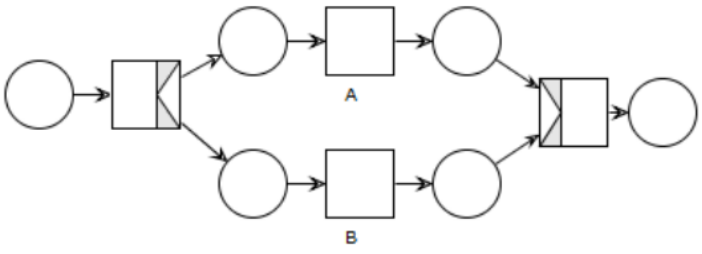
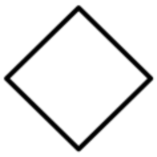
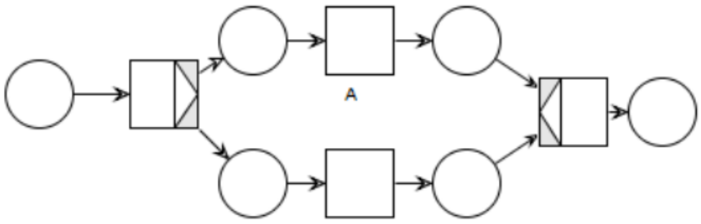
A workflow net is ***sound*** iff

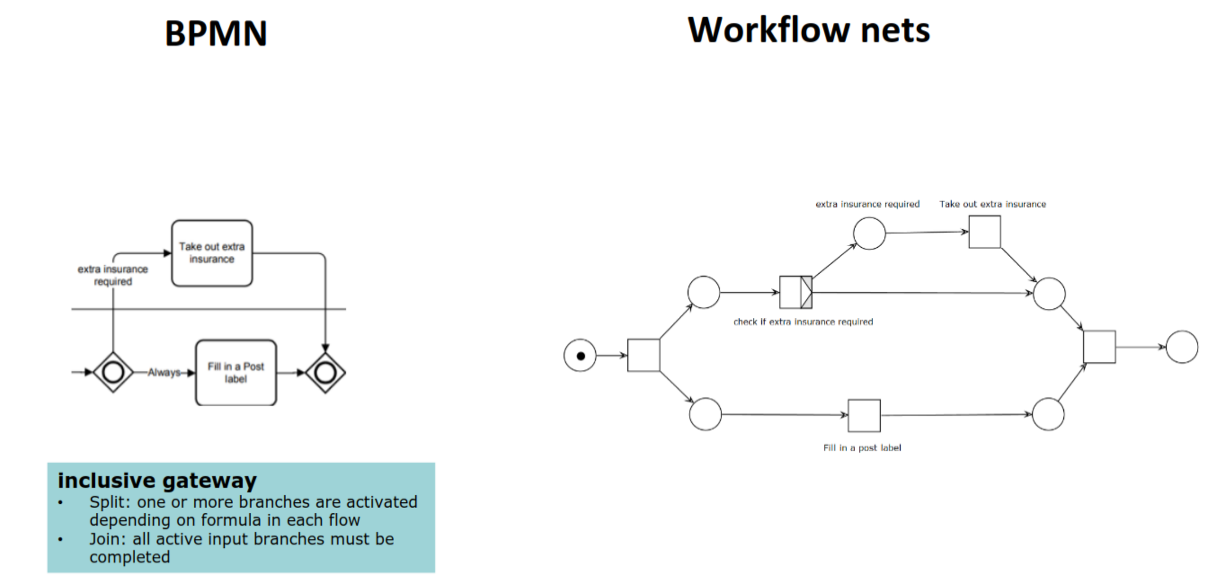
(1) every net execution starting from the *initial state* (one token in the source place, no tokens elsewhere) eventually leads to the *final state* (one token in the sink place, no tokens elsewhere), and

(2) every transition occurs in at least one net execution

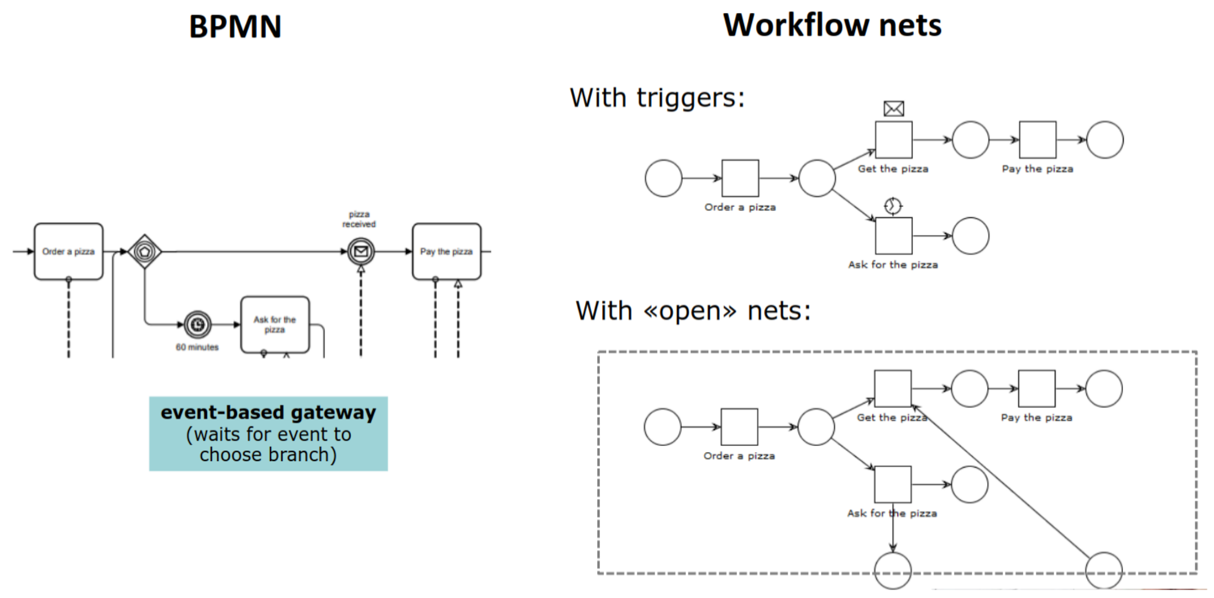
How to formally (and automatically) establish whether a net is sound?

A workflow net N is sound if and only if `(Nt, {i})` is live and boounded, where `Nt` is `N` extened witha transition from the sink place (end point) `o` to the source place `i`.

BPMN	Workflow nets
	
	
	
	
	



inclusive gateway
- Split: one or more branches are activated depending on formula in each flow
- Join: all active input branches must be completed



event-based gateway (waits for event to choose branch)

Steps

TODO: No particular steps to describe here -> need an update after lab second execution

Execution note

The program is in: `home/luca/WoPED/`. Right click on `Woped-Started.jar` and Select `Run with other program` -> Select `JAVA Runtime 17`

Lab 12: Functional test with `pytest` and Performance test with `locust`

Exam requirements:

Unit and Performance Testing

- Write and run unit tests with the pytest module, also computing coverage.
- Write and run locustfile.py to perform load testing.

Some test examples:

- Extend a given testset for a specific microservice written in Flask and fix a bug.
- Write a new (simple) microservice for the microase application and 3 tests for such a service.
- Launch an application with Docker [Compose] and write a locustfile.py to perform load testing against it. Specify that an endpoint is called 2 times more than others.

Unit test with pytest on `math_py` service

1. `pip install pytest`
2. Download the test-lab.zip primer
3. Add files like `a_test.py` to unit test the `math_py` microservice.
4. Use tests to spot and fix bugs. Note:
   - `conftest.py` configures the mock microservice: the following snippet unify tests and conftest.py in a single file.

```
####################### a_test.py
import pytest

from multistring.app import create_app

@pytest.fixture()
def app():
    app = create_app()
    app.config.update({
        "TESTING": True,
    })
    yield app

################################## OFFICIAL SOLUTION
```

```
def test_base(app):
    client = app.test_client()
    r = client.get('/multistring?a=Hello&n=3')
    assert r.status_code == 200
    assert r.json == {'s': 'HelloHelloHello'}

def test_negative(app):
    client = app.test_client()
    r = client.get('/multistring?a=Hello&n=-3')
    assert r.status_code == 400

def test_zero(app):
    client = app.test_client()
    r = client.get('/multistring?a=Hello&n=0')
    assert r.status_code == 200
    assert r.json == {'s': ''}

def test_empty(app):
    client = app.test_client()
    r = client.get('/multistring?a=&n=1')
    assert r.status_code == 200
    assert r.json == {'s': ''}

def test_missing(app):
    client = app.test_client()
    r = client.get('/multistring')
    assert r.status_code == 400
```

- `app.py` has been modified to inhibit `update_stats(service, op)` in testing mode. Here an example to add a `mulstring` endpoint.

```
from flask import Flask, render_template, request, make_response, jsonify
import requests
import time
import sys

app = Flask(__name__, instance_relative_config=True)

@app.route('/mulstring')
def mulstring():
    a = request.args.get('a', type=str)
    n = request.args.get('n', type=int)
    if a is not None and n is not None:
        if n<0:
            return make_response('Invalid input [n<0]\n', 400)
        return make_response(jsonify(s=a*n), 200)
    else:
        return make_response('Invalid input [a or n]\n', 400)

def create_app():
    return app
```

```
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

5. Execute `python3 -m pytest`
6. Evaluate test coverage by installing `pip install pytest-cov` and execute `python -m pytest --cov=[SERVICE-FOLDER-NAME]`

Load test with Locust

1. Install via `pip install locust`
2. Create `locustfile.py` in your root project folder to define users behaviours. For `microase` was:

```
import time
from locust import HttpUser, task, between


class QuickstartUser(HttpUser):
    wait_time = between(1, 2)

    @task
    def index_page(self):
        self.client.get("/")
        self.client.get("/")

    @task(3)#3 times more likely to be chosen than other tasksdef upper(self):
        for _ in range(10):
            self.client.get(f"/str/upper?a=aaaa", name="/str/upper")
            time.sleep(1)

    @task
    def mul(self):
        for a in range(10):
            self.client.get(f"/math/mul?a={a}00&b=9999", name="/math/mul")
            time.sleep(1)

    @task
    def stats(self):
        for _ in range(10):
            self.client.get(f"/stats", name="/stats")
            time.sleep(1)
```

1. Start the service: for `microase` execute `docker compose build` and `docker compose up`
2. Execute `locust` on new terminal. (**Luca-PC:** *$PATH* does not see the binary, use **`~/.local/bin/locust`** inside the folder with `locust.py`, porcamadonna)
3. Browse http://localhost:8089

4. Set parameters for locust swarm

NOTE: Assuming there is a bottleneck service e.g. `string-service`, you can scale it by `docker-compose up -d --scale string-service=6 --no-recreate`