Architecture Selenium Grid + N8N pour Scraping en Temps Réel

Projet : Automatisation des scripts Selenium pour BMP Amadeus
Date : 31 Janvier 2026
Version : 1.0

Table des matières

Contexte et Objectifs

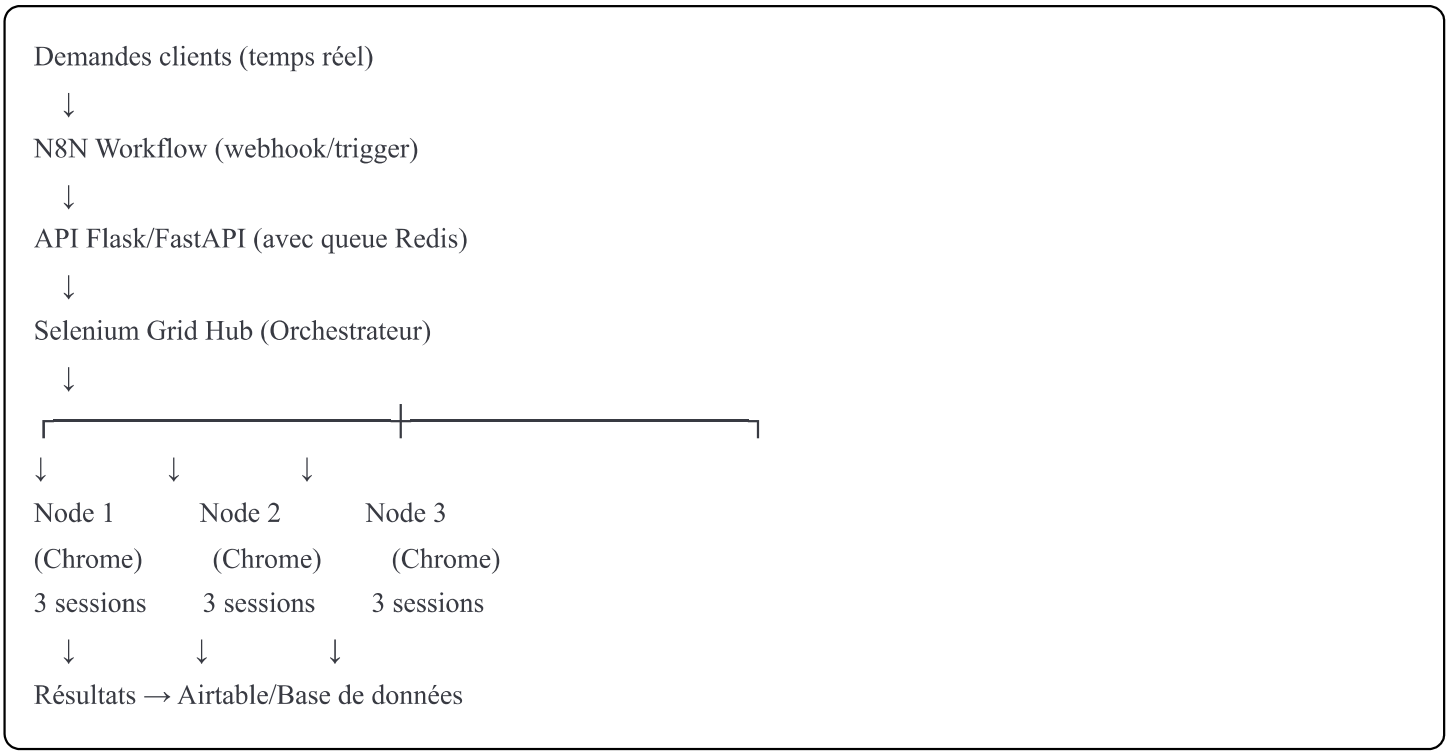
Situation actuelle

- **~10 scripts Selenium** actifs
- Demandes clients en **pseudo temps réel**
- Setup actuel : API Flask → Scripts Python/Selenium locaux
- **Problème** : Goulot d'étranglement, un seul script à la fois

Objectifs

- ✅ Exécuter **plusieurs scripts simultanément** (jusqu'à 9 en parallèle)
- ✅ Répondre rapidement aux **demandes clients temps réel**
- ✅ **Scalabilité** pour montée en charge
- ✅ **Isolation** des sessions (crash d'un script n'affecte pas les autres)
- ✅ Intégration transparente avec **N8N**

Architecture Globale



Composants principaux

Composant	Rôle	Port
Selenium Hub	Orchestrateur central, dispatche les requêtes	4444
Chrome Nodes	Workers exécutant les scripts (3 nodes × 3 sessions)	-
Redis	Queue de jobs, gestion asynchrone	6379
API Flask	Interface REST pour N8N et clients	8000
RQ Worker	Traite les jobs en background	-

Qu'est-ce que Selenium Grid ?

Définition

Selenium Grid est une **infrastructure distribuée** permettant d'exécuter des tests/scripts Selenium sur **plusieurs navigateurs et machines simultanément**.

Fonctionnement

1. **Client** envoie une requête WebDriver au **Hub**
2. **Hub** trouve un **Node** disponible avec les capacités demandées

3. **Node** exécute le script dans un navigateur isolé
4. Résultats renvoyés au client via le Hub

Avantages vs Setup actuel

Critère	Setup actuel	Avec Selenium Grid
Scripts simultanés	1	9 (3 nodes × 3 sessions)
Temps d'attente	Sériel (bloquant)	Parallèle (non-bloquant)
Isolation	Aucune	Totale (conteneurs)
Scalabilité	Limitée	Horizontale (ajouter nodes)
Robustesse	Un crash = tout bloque	Crashes isolés
Complexité	Simple	Moyenne
Ressources	Faibles	Moyennes/Élevées

Configuration Docker

Architecture des conteneurs

```
selenium-grid-network
├── selenium-hub (orchestrateur)
├── chrome-node-1 (3 sessions max)
├── chrome-node-2 (3 sessions max)
├── chrome-node-3 (3 sessions max)
├── redis (queue)
├── api-selenium (API Flask)
└── rq-worker (traite jobs)
```

docker-compose.yml

yaml

```yaml
version: '3.8'

services:
  # ==================================
  # Selenium Hub (Orchestrateur)
  # ==================================
  selenium-hub:
    image: selenium/hub:4.16.1
    container_name: selenium-hub
    ports:
      - "4444:4444"      # Web interface et API
      - "4442:4442"      # Event bus publish
      - "4443:4443"      # Event bus subscribe
    environment:
      - SE_SESSION_REQUEST_TIMEOUT=300
      - SE_SESSION_RETRY_INTERVAL=5
      - SE_HEALTHCHECK_INTERVAL=120
      - GRID_MAX_SESSION=10
      - GRID_BROWSER_TIMEOUT=300
      - GRID_TIMEOUT=300
    restart: unless-stopped
    networks:
      - selenium-grid

  # ==================================
  # Chrome Nodes (Workers)
  # ==================================
  chrome-node-1:
    image: selenium/node-chrome:4.16.1
    container_name: chrome-node-1
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=3
      - SE_NODE_SESSION_TIMEOUT=300
      - SE_SCREEN_WIDTH=1920
      - SE_SCREEN_HEIGHT=1080
    shm_size: 2gb
    volumes:
      - /dev/shm:/dev/shm
      - ./downloads:/home/seluser/Downloads
    restart: unless-stopped
    networks:
```

```
    - selenium-grid

  chrome-node-2:
    image: selenium/node-chrome:4.16.1
    container_name: chrome-node-2
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=3
      - SE_NODE_SESSION_TIMEOUT=300
    shm_size: 2gb
    volumes:
      - /dev/shm:/dev/shm
      - ./downloads:/home/seluser/Downloads
    restart: unless-stopped
    networks:
      - selenium-grid

  chrome-node-3:
    image: selenium/node-chrome:4.16.1
    container_name: chrome-node-3
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
      - SE_NODE_MAX_SESSIONS=3
      - SE_NODE_SESSION_TIMEOUT=300
    shm_size: 2gb
    volumes:
      - /dev/shm:/dev/shm
      - ./downloads:/home/seluser/Downloads
    restart: unless-stopped
    networks:
      - selenium-grid

  # ================================
  # Redis (Queue)
  # ================================
  redis:
    image: redis:7-alpine
    container_name: selenium-redis
    ports:
```

```
    - "6379:6379"
  restart: unless-stopped
  volumes:
    - redis-data:/data
  networks:
    - selenium-grid

# ==================================
# API Flask
# ==================================
api-selenium:
  build: ./api
  container_name: api-selenium
  ports:
    - "8000:8000"
  environment:
    - SELENIUM_HUB_URL=http://selenium-hub:4444
    - REDIS_URL=redis://redis:6379
    - AIRTABLE_TOKEN=${AIRTABLE_TOKEN}
    - AIRTABLE_BASE_ID=${AIRTABLE_BASE_ID}
  depends_on:
    - selenium-hub
    - redis
  volumes:
    - ./scripts:/app/scripts
    - ./logs:/app/logs
    - ./downloads:/app/downloads
  restart: unless-stopped
  networks:
    - selenium-grid

# ==================================
# RQ Worker (Traite les jobs)
# ==================================
rq-worker:
  build: ./api
  container_name: rq-worker
  command: python worker.py
  environment:
    - SELENIUM_HUB_URL=http://selenium-hub:4444
    - REDIS_URL=redis://redis:6379
    - AIRTABLE_TOKEN=${AIRTABLE_TOKEN}
  depends_on:
    - redis
    - selenium-hub
  volumes:
    - ./scripts:/app/scripts
```

```
    - ./logs:/app/logs
    restart: unless-stopped
    networks:
      - selenium-grid

networks:
  selenium-grid:
    driver: bridge

volumes:
  redis-data:
```

Fichiers de configuration supplémentaires

.env

```
env

AIRTABLE_TOKEN=votre_token_airtable
AIRTABLE_BASE_ID=appqkDWbZXhL0f7ge
```

API Flask avec Queue Redis

Structure du projet

```
selenium-grid-project/
├── docker-compose.yml
├── .env
├── api/
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── app.py
│   ├── worker.py
│   └── config.py
├── scripts/
│   ├── __init__.py
│   ├── scrape_bmp.py
│   ├── scrape_amadeus.py
│   └── ... (autres scripts)
├── logs/
└── downloads/
```

api/app.py

```python
```

```python
from flask import Flask, request, jsonify
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from redis import Redis
from rq import Queue
import logging
import json
from datetime import datetime
import os

app = Flask(__name__)

# Configuration
SELENIUM_HUB = os.getenv('SELENIUM_HUB_URL', 'http://selenium-hub:4444/wd/hub')
REDIS_URL = os.getenv('REDIS_URL', 'redis://redis:6379')
redis_conn = Redis.from_url(REDIS_URL)
queue = Queue('selenium_tasks', connection=redis_conn)

# Logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('/app/logs/api.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

def get_chrome_options():
    """Options Chrome optimisées pour headless"""
    chrome_options = Options()
    chrome_options.add_argument('--headless')
    chrome_options.add_argument('--no-sandbox')
    chrome_options.add_argument('--disable-dev-shm-usage')
    chrome_options.add_argument('--disable-gpu')
    chrome_options.add_argument('--window-size=1920,1080')
    chrome_options.add_argument('--disable-blink-features=AutomationControlled')
    chrome_options.add_experimental_option("excludeSwitches", ["enable-automation"])
    chrome_options.add_experimental_option('useAutomationExtension', False)

    # User agent pour éviter détection
    chrome_options.add_argument('user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36')

    return chrome_options
```

```python
def execute_selenium_script(script_name, parameters):
    """
    Fonction worker qui exécute le script Selenium
    Appelée de manière asynchrone par RQ
    """
    driver = None
    start_time = datetime.now()

    try:
        logger.info(f"Démarrage script: {script_name} avec params: {parameters}")

        # Connexion au Grid
        driver = webdriver.Remote(
            command_executor=SELENIUM_HUB,
            options=get_chrome_options()
        )

        # Import dynamique du script
        script_module = __import__(f'scripts.{script_name}', fromlist=['run'])

        # Exécution
        result = script_module.run(driver, parameters)

        execution_time = (datetime.now() - start_time).total_seconds()

        logger.info(f"Script {script_name} terminé en {execution_time}s")

        return {
            "status": "success",
            "script": script_name,
            "execution_time": execution_time,
            "data": result,
            "timestamp": datetime.now().isoformat()
        }

    except Exception as e:
        logger.error(f"Erreur script {script_name}: {str(e)}", exc_info=True)
        return {
            "status": "error",
            "script": script_name,
            "error": str(e),
            "timestamp": datetime.now().isoformat()
        }

    finally:
        if driver:
            try:
```

```
            driver.quit()
        except:
            pass

@app.route('/health', methods=['GET'])
def health_check():
    """Vérification de l'état du système"""
    try:
        # Vérifier connexion Redis
        redis_conn.ping()

        # Vérifier Grid Hub
        import requests
        hub_url = SELENIUM_HUB.replace('/wd/hub', '')
        hub_response = requests.get(f"{hub_url}/status", timeout=5)
        hub_status = hub_response.json()

        return jsonify({
            "status": "healthy",
            "redis": "connected",
            "selenium_grid": {
                "ready": hub_status['value']['ready'],
                "nodes": len(hub_status['value']['nodes']),
                "max_sessions": sum(node['maxSessions'] for node in hub_status['value']['nodes'])
            },
            "queue_size": len(queue),
            "timestamp": datetime.now().isoformat()
        })
    except Exception as e:
        logger.error(f"Health check failed: {str(e)}")
        return jsonify({
            "status": "unhealthy",
            "error": str(e),
            "timestamp": datetime.now().isoformat()
        }), 500

@app.route('/execute', methods=['POST'])
def execute_script():
    """
    Endpoint pour exécuter un script Selenium de manière synchrone
    Attend le résultat avant de répondre
    """
    data = request.json
    script_name = data.get('script')
    parameters = data.get('parameters', {})

    if not script_name:
```

```
        return jsonify({"error": "Script name required"}), 400

    try:
        result = execute_selenium_script(script_name, parameters)
        return jsonify(result)
    except Exception as e:
        logger.error(f"Erreur exécution synchrone: {str(e)}")
        return jsonify({
            "status": "error",
            "error": str(e)
        }), 500

@app.route('/execute/async', methods=['POST'])
def execute_script_async():
    """
    Endpoint pour exécuter un script Selenium de manière asynchrone
    Retourne immédiatement un job_id
    """
    data = request.json
    script_name = data.get('script')
    parameters = data.get('parameters', {})
    priority = data.get('priority', 'normal')  # high, normal, low

    if not script_name:
        return jsonify({"error": "Script name required"}), 400

    try:
        # Mise en queue avec timeout
        job = queue.enqueue(
            execute_selenium_script,
            script_name,
            parameters,
            job_timeout='10m',
            result_ttl=3600  # Garder résultats 1h
        )

        logger.info(f"Job {job.id} créé pour script {script_name}")

        return jsonify({
            "status": "queued",
            "job_id": job.id,
            "script": script_name,
            "position_in_queue": len(queue),
            "timestamp": datetime.now().isoformat()
        }), 202

    except Exception as e:
```

```
        logger.error(f"Erreur mise en queue: {str(e)}")
        return jsonify({
            "status": "error",
            "error": str(e)
        }), 500

@app.route('/job/<job_id>', methods=['GET'])
def get_job_status(job_id):
    """Récupérer le statut et résultat d'un job"""
    try:
        from rq.job import Job
        job = Job.fetch(job_id, connection=redis_conn)

        response = {
            "job_id": job.id,
            "status": job.get_status(),
            "created_at": job.created_at.isoformat() if job.created_at else None,
            "started_at": job.started_at.isoformat() if job.started_at else None,
            "ended_at": job.ended_at.isoformat() if job.ended_at else None,
        }

        if job.is_finished:
            response["result"] = job.result
        elif job.is_failed:
            response["error"] = str(job.exc_info)

        return jsonify(response)

    except Exception as e:
        return jsonify({
            "error": f"Job not found: {str(e)}"
        }), 404

@app.route('/queue/status', methods=['GET'])
def queue_status():
    """Statut détaillé de la queue"""
    from rq.registry import StartedJobRegistry, FinishedJobRegistry, FailedJobRegistry

    started_registry = StartedJobRegistry('selenium_tasks', connection=redis_conn)
    finished_registry = FinishedJobRegistry('selenium_tasks', connection=redis_conn)
    failed_registry = FailedJobRegistry('selenium_tasks', connection=redis_conn)

    return jsonify({
        "queued": len(queue),
        "running": len(started_registry),
        "finished": len(finished_registry),
        "failed": len(failed_registry),
```

```
        "timestamp": datetime.now().isoformat()
    })

@app.route('/scripts', methods=['GET'])
def list_scripts():
    """Liste les scripts Selenium disponibles"""
    import os
    scripts_dir = '/app/scripts'
    scripts = []

    for file in os.listdir(scripts_dir):
        if file.endswith('.py') and not file.startswith('__'):
            script_name = file.replace('.py', '')
            scripts.append({
                "name": script_name,
                "file": file
            })

    return jsonify({
        "scripts": scripts,
        "count": len(scripts),
        "timestamp": datetime.now().isoformat()
    })

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000, debug=False)
```

api/worker.py

python

```
from redis import Redis
from rq import Worker, Queue, Connection
import os
import logging

# Configuration
REDIS_URL = os.getenv('REDIS_URL', 'redis://redis:6379')
redis_conn = Redis.from_url(REDIS_URL)

# Logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('/app/logs/worker.log'),
        logging.StreamHandler()
    ]
)

if __name__ == '__main__':
    with Connection(redis_conn):
        worker = Worker(['selenium_tasks'])
        worker.work()
```

api/requirements.txt

```
Flask==3.0.0
selenium==4.16.0
redis==5.0.1
rq==1.15.1
gunicorn==21.2.0
requests==2.31.0
pyairtable==2.3.3
python-dotenv==1.0.0
```

api/Dockerfile

```
dockerfile
```

```
FROM python:3.11-slim

WORKDIR /app

# Dépendances système
RUN apt-get update && apt-get install -y \
    gcc \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Dépendances Python
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Code
COPY . .

# Créer répertoires
RUN mkdir -p /app/logs /app/downloads /app/scripts

# Healthcheck
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

EXPOSE 8000

# Gunicorn en production
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "--workers", "4", "--timeout", "300", "app:app"]
```

Scripts Selenium Modulaires

Structure d'un script

Chaque script doit avoir une fonction `run(driver, parameters)` qui :

- Reçoit une instance WebDriver déjà connectée au Grid
- Reçoit des paramètres sous forme de dict
- Retourne les données extraites
- Lève une exception en cas d'erreur

scripts/scrape_bmp.py

```
python
```

```
"""
Script de scraping BMP Amadeus
"""
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException
import time
import logging

logger = logging.getLogger(__name__)

def run(driver, parameters):
    """
    Script de scraping BMP Amadeus

    Parameters:
        driver (WebDriver): Instance WebDriver connectée au Grid
        parameters (dict): {
            'username': str,
            'password': str,
            'date_from': str (YYYY-MM-DD),
            'date_to': str (YYYY-MM-DD),
            'agency_code': str (optionnel)
        }

    Returns:
        dict: {
            'records': list,
            'count': int,
            'date_range': str
        }
    """

    # Extraction des paramètres
    username = parameters.get('username')
    password = parameters.get('password')
    date_from = parameters.get('date_from')
    date_to = parameters.get('date_to')
    agency_code = parameters.get('agency_code', '')

    if not all([username, password, date_from, date_to]):
        raise ValueError("Paramètres manquants: username, password, date_from, date_to requis")

    try:
        logger.info(f"Navigation vers BMP Amadeus...")
```

```
driver.get('https://bmp.viaamadeus.com/')

# Attendre le chargement de la page
wait = WebDriverWait(driver, 15)

# ================================
# AUTHENTIFICATION
# ================================
logger.info(f"Authentification pour {username}...")

username_field = wait.until(
    EC.presence_of_element_located((By.ID, "username"))
)
username_field.clear()
username_field.send_keys(username)

password_field = driver.find_element(By.ID, "password")
password_field.clear()
password_field.send_keys(password)

login_button = driver.find_element(By.CSS_SELECTOR, "button[type='submit']")
login_button.click()

# Attendre le chargement du dashboard
logger.info("Attente du dashboard...")
wait.until(
    EC.presence_of_element_located((By.ID, "dashboard"))
)

time.sleep(2)  # Attendre stabilisation

# ================================
# NAVIGATION VERS RAPPORTS
# ================================
logger.info("Navigation vers les rapports...")

reports_menu = driver.find_element(By.LINK_TEXT, "Rapports")
reports_menu.click()

time.sleep(1)

transactions_link = wait.until(
    EC.element_to_be_clickable((By.LINK_TEXT, "Transactions"))
)
transactions_link.click()

# ================================
```

```
# FILTRAGE PAR DATES
# ==============================
logger.info(f"Filtrage par dates: {date_from} à {date_to}...")

date_from_field = wait.until(
    EC.presence_of_element_located((By.ID, "date_from"))
)
date_from_field.clear()
date_from_field.send_keys(date_from)

date_to_field = driver.find_element(By.ID, "date_to")
date_to_field.clear()
date_to_field.send_keys(date_to)

# Agence si spécifiée
if agency_code:
    agency_field = driver.find_element(By.ID, "agency_code")
    agency_field.clear()
    agency_field.send_keys(agency_code)

# Lancer la recherche
search_button = driver.find_element(By.ID, "search_button")
search_button.click()

# Attendre les résultats
logger.info("Attente des résultats...")
wait.until(
    EC.presence_of_element_located((By.CSS_SELECTOR, "table.transactions-table"))
)

time.sleep(2)  # Attendre chargement complet

# ==============================
# EXTRACTION DES DONNÉES
# ==============================
logger.info("Extraction des données...")

data = []
rows = driver.find_elements(By.CSS_SELECTOR, "table.transactions-table tbody tr")

for row in rows:
    try:
        cols = row.find_elements(By.TAG_NAME, "td")

        if len(cols) >= 6:
            record = {
                "date": cols[0].text.strip(),
```

```
                "reference": cols[1].text.strip(),
                "agence": cols[2].text.strip(),
                "type": cols[3].text.strip(),
                "montant": cols[4].text.strip(),
                "statut": cols[5].text.strip(),
            }
            data.append(record)
        except Exception as e:
            logger.warning(f"Erreur extraction ligne: {str(e)}")
            continue

    logger.info(f"Extraction terminée: {len(data)} enregistrements")

    # ================================
    # DÉCONNEXION
    # ================================
    try:
        logout_button = driver.find_element(By.ID, "logout")
        logout_button.click()
        time.sleep(1)
    except:
        logger.warning("Impossible de se déconnecter proprement")

    return {
        "records": data,
        "count": len(data),
        "date_range": f"{date_from} à {date_to}",
        "agency": agency_code if agency_code else "Toutes"
    }

except TimeoutException as e:
    raise Exception(f"Timeout: {str(e)}")
except Exception as e:
    # Capturer screenshot en cas d'erreur
    try:
        screenshot_path = f"/app/logs/error_{int(time.time())}.png"
        driver.save_screenshot(screenshot_path)
        logger.error(f"Screenshot sauvegardé: {screenshot_path}")
    except:
        pass

    raise Exception(f"Erreur scraping BMP: {str(e)}")
```

scripts/init.py

```
python
```

```
# Permet l'import dynamique des scripts
```

Template pour nouveau script

python

```
"""
Script: [NOM_DU_SCRIPT]
Description: [DESCRIPTION]
"""
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import logging

logger = logging.getLogger(__name__)

def run(driver, parameters):
    """
    [Description détaillée]

    Parameters:
        driver (WebDriver): Instance WebDriver
        parameters (dict): {
            'param1': description,
            'param2': description
        }

    Returns:
        dict: Données extraites
    """

    try:
        logger.info(f"Démarrage script...")

        # Votre logique ici
        driver.get('https://example.com')

        # ...

        return {
            "status": "success",
            "data": []
        }

    except Exception as e:
        raise Exception(f"Erreur: {str(e)}")
```

Intégration N8N

Workflow 1 : Exécution synchrone (attendre résultat)

Cas d'usage : Requête client unique, besoin du résultat immédiat

```json
```

```
{
 "name": "Scrape BMP - Synchrone",
 "nodes": [
  {
   "parameters": {
    "httpMethod": "POST",
    "path": "scrape-bmp",
    "responseMode": "responseNode",
    "options": {}
   },
   "name": "Webhook",
   "type": "n8n-nodes-base.webhook",
   "position": [250, 300]
  },
  {
   "parameters": {
    "url": "http://api-selenium:8000/execute",
    "authentication": "genericCredentialType",
    "genericAuthType": "httpHeaderAuth",
    "sendBody": true,
    "bodyParameters": {
     "parameters": [
      {
       "name": "script",
       "value": "scrape_bmp"
      },
      {
       "name": "parameters",
       "value": "={{ { username: $json.username, password: $json.password, date_from: $json.date_from, date_to: $json.da
      }
     ]
    },
    "options": {
     "timeout": 300000
    }
   },
   "name": "Execute Selenium Script",
   "type": "n8n-nodes-base.httpRequest",
   "position": [450, 300]
  },
  {
   "parameters": {
    "operation": "appendOrUpdate",
    "baseId": "appqkDWbZXhL0f7ge",
    "tableId": "tblEDA8qjsNJjS7zc",
    "columns": "mappingMode",
```

```
      "fieldsUi": {
        "fieldValues": [
          {
            "fieldId": "fldDate",
            "fieldValue": "={{ $json.data.records[0].date }}"
          },
          {
            "fieldId": "fldReference",
            "fieldValue": "={{ $json.data.records[0].reference }}"
          }
        ]
      }
    },
    "name": "Save to Airtable",
    "type": "n8n-nodes-base.airtable",
    "position": [650, 300]
  },
  {
    "parameters": {
      "respondWith": "json",
      "responseBody": "={{ $json }}"
    },
    "name": "Respond to Webhook",
    "type": "n8n-nodes-base.respondToWebhook",
    "position": [850, 300]
  }
 ],
 "connections": {
  "Webhook": {
    "main": [[{"node": "Execute Selenium Script", "type": "main", "index": 0}]]
  },
  "Execute Selenium Script": {
    "main": [[{"node": "Save to Airtable", "type": "main", "index": 0}]]
  },
  "Save to Airtable": {
    "main": [[{"node": "Respond to Webhook", "type": "main", "index": 0}]]
  }
 }
}
```

Workflow 2 : Exécution asynchrone (non-bloquant)

Cas d'usage : Multiples requêtes simultanées, pas besoin d'attendre

```
json
```

```
{
 "name": "Scrape BMP - Asynchrone",
 "nodes": [
  {
   "parameters": {
    "httpMethod": "POST",
    "path": "scrape-bmp-async",
    "responseMode": "responseNode"
   },
   "name": "Webhook",
   "type": "n8n-nodes-base.webhook",
   "position": [250, 300]
  },
  {
   "parameters": {
    "url": "http://api-selenium:8000/execute/async",
    "sendBody": true,
    "bodyParameters": {
     "parameters": [
      {
       "name": "script",
       "value": "scrape_bmp"
      },
      {
       "name": "parameters",
       "value": "={{ $json }}"
      },
      {
       "name": "priority",
       "value": "high"
      }
     ]
    }
   },
   "name": "Queue Job",
   "type": "n8n-nodes-base.httpRequest",
   "position": [450, 300]
  },
  {
   "parameters": {
    "respondWith": "json",
    "responseBody": "={{ { job_id: $json.job_id, status: $json.status, message: 'Job en cours, consultez /job/' + $json.job_id
   },
   "name": "Return Job ID",
   "type": "n8n-nodes-base.respondToWebhook",
   "position": [650, 300]
```

```
    }
  ],
  "connections": {
    "Webhook": {
      "main": [[{"node": "Queue Job", "type": "main", "index": 0}]]
    },
    "Queue Job": {
      "main": [[{"node": "Return Job ID", "type": "main", "index": 0}]]
    }
  }
}
```

Workflow 3 : Polling du statut job

Pour récupérer les résultats d'un job asynchrone

json

```
{
 "name": "Check Job Status",
 "nodes": [
  {
   "parameters": {
    "httpMethod": "POST",
    "path": "check-job",
    "responseMode": "responseNode"
   },
   "name": "Webhook",
   "type": "n8n-nodes-base.webhook",
   "position": [250, 300]
  },
  {
   "parameters": {
    "url": "=http://api-selenium:8000/job/{{ $json.job_id }}",
    "options": {}
   },
   "name": "Get Job Status",
   "type": "n8n-nodes-base.httpRequest",
   "position": [450, 300]
  },
  {
   "parameters": {
    "conditions": {
     "string": [
      {
       "value1": "={{ $json.status }}",
       "value2": "finished"
      }
     ]
    }
   },
   "name": "Is Finished?",
   "type": "n8n-nodes-base.if",
   "position": [650, 300]
  },
  {
   "parameters": {
    "operation": "appendOrUpdate",
    "baseId": "appqkDWbZXhL0f7ge",
    "tableId": "tblEDA8qjsNJjS7zc"
   },
   "name": "Save Results",
   "type": "n8n-nodes-base.airtable",
   "position": [850, 200]
```

```
    },
    {
      "parameters": {
        "respondWith": "json",
        "responseBody": "={{ $json }}"
      },
      "name": "Return Status",
      "type": "n8n-nodes-base.respondToWebhook",
      "position": [850, 400]
    }
  ],
  "connections": {
    "Webhook": {
      "main": [[{"node": "Get Job Status", "type": "main", "index": 0}]]
    },
    "Get Job Status": {
      "main": [[{"node": "Is Finished?", "type": "main", "index": 0}]]
    },
    "Is Finished?": {
      "main": [
        [{"node": "Save Results", "type": "main", "index": 0}],
        [{"node": "Return Status", "type": "main", "index": 0}]
      ]
    }
  }
}
```

Workflow 4 : Planification automatique

Exécution automatique toutes les heures

```
json
```

```
{
 "name": "Scrape BMP - Scheduled",
 "nodes": [
  {
   "parameters": {
    "rule": {
     "interval": [
      {
       "field": "hours",
       "hoursInterval": 1
      }
     ]
    }
   },
   "name": "Schedule Trigger",
   "type": "n8n-nodes-base.scheduleTrigger",
   "position": [250, 300]
  },
  {
   "parameters": {
    "functionCode": "const today = new Date();\nconst yesterday = new Date(today);\nyesterday.setDate(yesterday.getDate(
   },
   "name": "Prepare Parameters",
   "type": "n8n-nodes-base.function",
   "position": [450, 300]
  },
  {
   "parameters": {
    "url": "http://api-selenium:8000/execute/async",
    "sendBody": true,
    "bodyParameters": {
     "parameters": [
      {
       "name": "script",
       "value": "scrape_bmp"
      },
      {
       "name": "parameters",
       "value": "={{ $json }}"
      }
     ]
    }
   },
   "name": "Execute Script",
   "type": "n8n-nodes-base.httpRequest",
   "position": [650, 300]
```

```
    }
  ],
  "connections": {
    "Schedule Trigger": {
      "main": [[{"node": "Prepare Parameters", "type": "main", "index": 0}]]
    },
    "Prepare Parameters": {
      "main": [[{"node": "Execute Script", "type": "main", "index": 0}]]
    }
  }
}
```

Plan de Migration

Phase 1 : Installation et tests (2-4 heures)

Étape 1.1 : Préparation serveur

```bash
# Se connecter au serveur
ssh user@votre-serveur

# Créer répertoire projet
mkdir -p /opt/selenium-grid
cd /opt/selenium-grid

# Vérifier Docker
docker --version
docker-compose --version

# Si pas installés
sudo yum install docker docker-compose  # CentOS/AlmaLinux
# ou
sudo apt install docker.io docker-compose  # Ubuntu
```

Étape 1.2 : Déploiement Grid

```bash
```

```bash
# Créer structure
mkdir -p api/scripts logs downloads

# Copier les fichiers de configuration
# - docker-compose.yml
# - api/Dockerfile
# - api/requirements.txt
# - api/app.py
# - api/worker.py
# - .env

# Configurer .env
nano .env
# Ajouter:
# AIRTABLE_TOKEN=votre_token
# AIRTABLE_BASE_ID=appqkDWbZXhL0f7ge

# Lancer les conteneurs
docker-compose up -d

# Vérifier logs
docker-compose logs -f
```

Étape 1.3 : Tests initiaux

```bash
```

```
# Test 1: Health check
curl http://localhost:8000/health

# Test 2: Interface Grid
# Naviguer vers http://votre-serveur:4444

# Test 3: Scripts disponibles
curl http://localhost:8000/scripts

# Test 4: Exécution test script
curl -X POST http://localhost:8000/execute \
  -H "Content-Type: application/json" \
  -d '{
    "script": "scrape_bmp",
    "parameters": {
      "username": "test",
      "password": "test",
      "date_from": "2026-01-01",
      "date_to": "2026-01-31"
    }
  }'
```

Phase 2 : Migration scripts (4-6 heures)

Adaptation des scripts existants

Pour chaque script :

1. **Créer fichier dans scripts/**

```
python

  # scripts/mon_script.py
  def run(driver, parameters):
      # Votre logique existante
      pass
```

2. **Adapter le code**
   - Remplacer `webdriver.Chrome()` par utilisation du driver passé en param
   - Retirer initialisation/fermeture driver (géré par l'API)
   - Structurer les retours en dict
3. **Tester individuellement**

```
bash
```

```
curl -X POST http://localhost:8000/execute \
  -H "Content-Type: application/json" \
  -d '{"script": "mon_script", "parameters": {...}}'
```

Scripts à migrer (exemple)

```
✓ scrape_bmp.py
✓ scrape_amadeus_booking.py
✓ scrape_transactions.py
✓ extract_reports.py
✓ ... (6 autres scripts)
```

Phase 3 : Intégration N8N (2-3 heures)

Modification des workflows

1. **Identifier workflows utilisant Selenium**
   - Lister tous les workflows N8N concernés
   - Noter les endpoints actuels
2. **Créer nouveaux workflows**
   - Dupliquer workflows existants
   - Modifier endpoints vers nouvelle API
   - Tester en parallèle
3. **Validation**
   - Comparer résultats ancien vs nouveau système
   - Vérifier temps d'exécution
   - Valider données dans Airtable

Phase 4 : Bascule progressive (1-2 jours)

Jour 1 : Mode hybride

```
Trafic:
├── 20% → Nouveau système (Grid)
└── 80% → Ancien système
```

Monitoring intensif :

- Logs API : `docker logs -f api-selenium`
- Logs Worker : `docker logs -f rq-worker`
- Dashboard Grid : http://serveur:4444

- Queue Redis : `curl http://localhost:8000/queue/status`

Jour 2 : Montée en charge

```
Trafic:
├── 50% → Nouveau système
└── 50% → Ancien système
```

Ajustements si nécessaire :

- Ajouter nodes si saturation
- Tuning timeouts
- Optimisation scripts

Jour 3 : Bascule complète

```
Trafic:
└── 100% → Nouveau système (Grid)
```

Désactivation ancien système :

- Arrêter ancien service
- Garder en backup 1 semaine
- Supprimer après validation

Phase 5 : Optimisation continue

Semaine 1-2

- **Monitoring performances**
  - Temps moyen par script
  - Taux de succès/échec
  - Utilisation ressources
- **Ajustements**
  - Nombre de nodes
  - Sessions par node
  - Timeouts

Semaine 3-4

- **Scalabilité**
  - Ajouter node 4 si nécessaire

- Tests de charge
- Plan de reprise après incident

Monitoring et Maintenance

Dashboard de monitoring

Grafana + Prometheus (optionnel mais recommandé)

Ajouter au `docker-compose.yml` :

```yaml
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    ports:
      - "9090:9090"
    networks:
      - selenium-grid

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana-data:/var/lib/grafana
    networks:
      - selenium-grid
```

Alertes

Script de monitoring (monitoring.py)

```python
```

```python
import requests
import time
import smtplib
from email.mime.text import MIMEText

API_URL = "http://localhost:8000"
ALERT_EMAIL = "admin@votre-domaine.com"

def send_alert(subject, message):
    """Envoyer une alerte par email"""
    msg = MIMEText(message)
    msg['Subject'] = f"[SELENIUM GRID] {subject}"
    msg['From'] = "monitoring@votre-domaine.com"
    msg['To'] = ALERT_EMAIL

    # Configuration SMTP
    with smtplib.SMTP('localhost') as server:
        server.send_message(msg)

def check_system_health():
    """Vérifier l'état du système"""
    try:
        # Health check
        response = requests.get(f"{API_URL}/health", timeout=10)
        data = response.json()

        if data['status'] != 'healthy':
            send_alert("System Unhealthy", f"Status: {data}")
            return False

        # Vérifier queue
        queue_response = requests.get(f"{API_URL}/queue/status", timeout=10)
        queue_data = queue_response.json()

        # Alerte si queue saturée
        if queue_data['queued'] > 50:
            send_alert(
                "Queue Saturated",
                f"Queue size: {queue_data['queued']}, Running: {queue_data['running']}"
            )

        # Alerte si trop d'échecs
        if queue_data['failed'] > 10:
            send_alert(
                "High Failure Rate",
                f"Failed jobs: {queue_data['failed']}"
```

```
            )

        return True

    except Exception as e:
        send_alert("Monitoring Failed", str(e))
        return False

if __name__ == '__main__':
    while True:
        check_system_health()
        time.sleep(60)  # Vérifier toutes les minutes
```

Logs

Centralisation des logs

```bash
# Logs API
docker logs -f api-selenium > /var/log/selenium/api.log

# Logs Worker
docker logs -f rq-worker > /var/log/selenium/worker.log

# Logs Hub
docker logs -f selenium-hub > /var/log/selenium/hub.log
```

Rotation des logs

```bash
# /etc/logrotate.d/selenium-grid
/var/log/selenium/*.log {
    daily
    rotate 7
    compress
    delaycompress
    missingok
    notifempty
}
```

Maintenance régulière

Quotidienne

```bash
```

```
# Nettoyer conteneurs arrêtés
docker system prune -f

# Vérifier espace disque
df -h /var/lib/docker
```

Hebdomadaire

bash

```
# Redémarrer Grid (pendant heures creuses)
docker-compose restart

# Nettoyer logs anciens
find /app/logs -name "*.log" -mtime +7 -delete
```

Mensuelle

bash

```
# Mise à jour images Docker
docker-compose pull
docker-compose up -d

# Backup configuration
tar -czf backup-$(date +%Y%m%d).tar.gz \
  docker-compose.yml \
  api/ \
  scripts/ \
  .env
```

Backup et restauration

Backup automatique

bash

```bash
#!/bin/bash
# backup.sh

BACKUP_DIR="/backup/selenium-grid"
DATE=$(date +%Y%m%d_%H%M%S)

# Créer backup
mkdir -p $BACKUP_DIR
cd /opt/selenium-grid

tar -czf $BACKUP_DIR/selenium-grid-$DATE.tar.gz \
  docker-compose.yml \
  .env \
  api/ \
  scripts/

# Garder seulement 30 derniers jours
find $BACKUP_DIR -name "*.tar.gz" -mtime +30 -delete

echo "Backup completed: selenium-grid-$DATE.tar.gz"
```

Restauration

```bash
```

```bash
#!/bin/bash
# restore.sh

BACKUP_FILE=$1

if [ -z "$BACKUP_FILE" ]; then
  echo "Usage: ./restore.sh <backup-file.tar.gz>"
  exit 1
fi

# Arrêter services
cd /opt/selenium-grid
docker-compose down

# Restaurer
tar -xzf $BACKUP_FILE -C /opt/selenium-grid

# Redémarrer
docker-compose up -d

echo "Restoration completed"
```

Ressources Serveur

Configuration minimale

Ressource	Minimum	Recommandé	Optimal
CPU	4 cores	6 cores	8+ cores
RAM	8 GB	12 GB	16+ GB
Disk	20 GB	50 GB	100+ GB
Réseau	100 Mbps	1 Gbps	1 Gbps

Détail par service

Service	RAM	CPU	Notes
Selenium Hub	512 MB	0.5 core	Léger
Chrome Node (×3)	2 GB chacun	1 core chacun	Gourmand

Service	RAM	CPU	Notes
Redis	256 MB	0.2 core	Cache
API Flask	512 MB	1 core	Backend
RQ Worker	512 MB	1 core	Jobs
TOTAL	**~8 GB**	**~5 cores**	

Calcul capacité

Avec 3 nodes × 3 sessions = 9 scripts simultanés

Si un script prend en moyenne **2 minutes** :

- **Capacité horaire** : 9 × (60 / 2) = **270 scripts/heure**
- **Capacité journalière** : 270 × 24 = **6,480 scripts/jour**

Pour augmenter la capacité :

- Ajouter 1 node = +3 sessions = +30% capacité
- Ajouter 2 nodes = +6 sessions = +67% capacité

Scalabilité horizontale

```yaml
# Ajouter simplement plus de nodes dans docker-compose.yml

chrome-node-4:
  image: selenium/node-chrome:4.16.1
  container_name: chrome-node-4
  depends_on:
    - selenium-hub
  environment:
    - SE_EVENT_BUS_HOST=selenium-hub
    - SE_EVENT_BUS_PUBLISH_PORT=4442
    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
    - SE_NODE_MAX_SESSIONS=3
  shm_size: 2gb
  networks:
    - selenium-grid
```

Troubleshooting

Problème : Grid Hub ne démarre pas

Symptômes :

```
ERROR: Grid Hub container exiting
```

Solutions :

```bash
# Vérifier logs
docker logs selenium-hub

# Vérifier port 4444 disponible
sudo netstat -tlnp | grep 4444

# Redémarrer
docker-compose restart selenium-hub
```

Problème : Nodes ne se connectent pas au Hub

Symptômes :

```
Nodes: 0/3 connected
```

Solutions :

```bash
# Vérifier réseau Docker
docker network inspect selenium-grid_selenium-grid

# Vérifier logs nodes
docker logs chrome-node-1

# Redémarrer ensemble
docker-compose down
docker-compose up -d
```

Problème : Scripts timeout

Symptômes :

```
TimeoutException: Message: timeout
```

Solutions :

```python
# Augmenter timeouts dans script
wait = WebDriverWait(driver, 30)  # Au lieu de 15

# Ou dans docker-compose.yml
environment:
  - SE_NODE_SESSION_TIMEOUT=600  # 10 minutes
```

Problème : Out of Memory

Symptômes :

```
Container killed (OOM)
```

Solutions :

```bash
# Réduire sessions par node
environment:
  - SE_NODE_MAX_SESSIONS=2  # Au lieu de 3

# Ou augmenter RAM serveur
# Ou ajouter swap
sudo fallocate -l 4G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

Problème : Queue Redis saturée

Symptômes :

```
Queue size: 100+
```

Solutions :

```bash
```

```
# Ajouter worker
docker-compose up -d --scale rq-worker=3

# Ou augmenter priorité jobs
POST /execute/async
{
  "priority": "high"
}
```

Problème : Scripts échouent aléatoirement

Causes possibles :

1. Détection bot par le site
2. Timeouts réseau
3. Éléments page non chargés

Solutions :

```python
# 1. Anti-détection
chrome_options.add_argument('--disable-blink-features=AutomationControlled')
chrome_options.add_experimental_option("excludeSwitches", ["enable-automation"])

# 2. Attentes explicites
wait.until(EC.presence_of_element_located((By.ID, "element")))

# 3. Retry logic
from tenacity import retry, stop_after_attempt

@retry(stop=stop_after_attempt(3))
def scrape_with_retry():
    # Code scraping
    pass
```

Checklist de déploiement

Avant déploiement

- [ ] Serveur Linux prêt (CentOS/AlmaLinux/Ubuntu)
- [ ] Docker et Docker Compose installés
- [ ] Ports 4444, 8000, 6379 disponibles
- [ ] Ressources suffisantes (8GB RAM, 4 CPU min)

- [ ] Token Airtable configuré
- [ ] Credentials BMP Amadeus disponibles

Déploiement

- [ ] Structure répertoires créée
- [ ] Fichiers configuration copiés
- [ ] .env configuré avec tokens
- [ ] docker-compose.yml vérifié
- [ ] Scripts migrés dans /scripts
- [ ] Images Docker téléchargées
- [ ] Conteneurs démarrés
- [ ] Health check OK

Post-déploiement

- [ ] Test exécution script simple
- [ ] Test exécution parallèle (3+ scripts)
- [ ] Workflows N8N modifiés
- [ ] Test end-to-end depuis N8N
- [ ] Monitoring configuré
- [ ] Alertes configurées
- [ ] Backup automatique configuré
- [ ] Documentation équipe mise à jour

Validation finale

- [ ] Tous les scripts testés individuellement
- [ ] Charge normale supportée
- [ ] Pics de charge testés
- [ ] Temps réponse acceptable (<5min par script)
- [ ] Données correctement dans Airtable
- [ ] Logs accessibles et lisibles
- [ ] Procédure de restauration testée

Conclusion

Cette architecture **Selenium Grid + N8N** vous permettra de :

✅ **Scalabilité** : Passer de 1 à 9 scripts simultanés immédiatement
✅ **Fiabilité** : Isolation des sessions, auto-recovery
✅ **Performance** : Temps réponse optimal pour demandes temps réel
✅ **Maintenabilité** : Architecture modulaire, logs centralisés
✅ **Évolutivité** : Ajout facile de nodes selon charge

Prochaines étapes

1. **Phase pilote** : Déployer sur serveur de test
2. **Migration progressive** : 2-3 scripts en production
3. **Validation** : 1 semaine de monitoring
4. **Bascule complète** : Tous les scripts
5. **Optimisation** : Ajustements selon charge réelle

Support et ressources

- **Documentation Selenium Grid** : https://www.selenium.dev/documentation/grid/
- **Redis Queue (RQ)** : https://python-rq.org/
- **Flask** : https://flask.palletsprojects.com/
- **N8N** : https://docs.n8n.io/

Document préparé le 31 Janvier 2026
Version 1.0