Handout: Scaleway Kubernetes Kapsule with regional Private Networks and regional node pools

This use case demonstrates how to deploy a Scaleway Kubernetes Kapsule cluster integrated with a regional Private Network and node pools distributed across multiple Availability Zones (AZs) within a Scaleway region (e.g., `pl-waw`).

The goal is to create a **secure, scalable, and highly available environment** for a web application, ensuring **isolated network communication** and **fault tolerance**.
A **Node.js backend application** reads from and writes to a **PostgreSQL Managed Database**, and is exposed directly to the internet using a **Kubernetes LoadBalancer service**.
Sensitive database credentials are stored in a **Kubernetes Secret** created with `kubectl create secret generic`.
All resources are deployed and managed using the **Scaleway CLI** (`scw`) and **kubectl**.

Objectives

- **Security:** Use a Private Network to isolate communication between Kubernetes nodes, Managed Databases, and Load Balancers, minimizing exposure to the public internet. Database credentials are stored in a Kubernetes Secret.
- **High Availability**: Distribute node pools across multiple availability zones (e.g., pl-waw-1, pl-waw-2, pl-waw-3) to ensure resilience against AZ failures.
- **Scalability:** Enable autoscaling on node pools to dynamically adjust resources based on workload demands.
- **Easy management:** Use the Scaleway CLI and `kubectl` to streamline cluster creation, configuration, and secret management.

Prerequisites

- Scaleway CLI (`scw`) installed and configured with an API key.
- A Scaleway account with sufficient permissions to create Kubernetes clusters, Private Networks, Managed Databases, and Container Registry.
- You have created a Container Registry namespace
- Docker installed locally to build the Node.js application image.
- Basic familiarity with Kubernetes concepts (e.g., clusters, node pools, pods, secrets), Node.js.
- An existing Scaleway Project.

Creating a Private Network

1. Create a regional Private Network using `scw` to enable secure, isolated communication for the Kubernetes cluster and other resources. Make sure to replace <your-vpc-id> with the ID of the VPC (Virtual Private Cloud) you want to create the Private Network in.

```shell
scw vpc private-network create name=kapsule-private-network vpc-id=<your-vpc-id>
region=pl-waw
```

2. Note the `id` of the Private Network (e.g., `120b006c-7ede-4a62-9c18-d80057656670`) as it is required for the next steps.

Creating a Kubernetes Kapsule cluster

1. Create a Kapsule cluster in the `pl-waw` region, attaching it to the previously created Private Network and specifying a recent Kubernetes version (e.g., `1.32.3`).

```shell
scw k8s cluster create name=web-app-cluster \
  version=1.32.3 \
  cni=cilium \
  private-network-id=120b006c-7ede-4a62-9c18-d80057656670 \
  region=pl-waw
```

The cluster is configured with the Cilium CNI for networking and automatically assigned a /22 IP subnet for the Private Network.

2. Note the `id` of the cluster (e.g., `248ed689-5d57-418b-8208-179f87d251b0`), as it is required for the next steps.

Creating regional node pools

1. Create three node pools, each in a different Availability Zone (`pl-waw-1`, `pl-waw-2`, `pl-waw-3`), with autoscaling enabled to handle varying workloads. Use `PRO2-S` Instances for cost-effective and performant development.

Shell

```
# Node pool in pl-waw-1
scw k8s pool create \
  cluster-id=248ed689-5d57-418b-8208-179f87d251b0 \
  name=pool-pl-waw-1 \
  node-type=PRO2-S \
  size=2 \
  min-size=1 \
  max-size=5 \
  autoscaling=true \
  autohealing=true \
  zone=pl-waw-1

# Node pool in pl-waw-2
scw k8s pool create \
  cluster-id=248ed689-5d57-418b-8208-179f87d251b0 \
  name=pool-pl-waw-2 \
  node-type=PRO2-S \
  size=2 \
  min-size=1 \
  max-size=5 \
  autoscaling=true \
  autohealing=true \
  zone=pl-waw-2

# Node pool in pl-waw-3
scw k8s pool create \
  cluster-id=248ed689-5d57-418b-8208-179f87d251b0 \
  name=pool-pl-waw-3 \
  node-type=PRO2-S \
  size=2 \
  min-size=1 \
  max-size=5 \
  autoscaling=true \
  autohealing=true \
  zone=pl-waw-3
```

- Each pool starts with 2 nodes (`size=2`) but can scale between 1 (`min-size`) and 5 (`max-size`) nodes based on demand.

- `autoscaling=true` enables dynamic scaling, and `autohealing=true` ensures nodes are restarted or replaced if they become unresponsive for over 15 minutes.
- Specifying `zone` places each pool in a different AZ for multi-AZ resilience.

Creating a Managed Database

1. Create a PostgreSQL Managed Database instance in the `pl-waw` region.

```
Shell
scw rdb instance create node-type=DB-PLAY2-PICO \
  engine=PostgreSQL-16 \
  region=pl-waw \
  name=web-app-db \
  user-name=web-app-user \
  password=secret_Pa$$word_99 \
  volume-type=sbs_5k \
  volume-size=20GB
```

2. Take note of the instance `id` (e.g., `c39e632d-c6c9-4cf4-adcc-0827143f9e1f`) as it is required for the next step.

3. Create a new endpoint for the Managed Database instance to attach it to the Private Network.

```
Shell
scw rdb endpoint create instance-id=c39e632d-c6c9-4cf4-adcc-0827143f9e1f
private-network.private-network-id=120b006c-7ede-4a62-9c18-d80057656670
region=pl-waw
```

4. Create a new database for the application.

```
Shell
scw rdb database create instance-id=c39e632d-c6c9-4cf4-adcc-0827143f9e1f
name=web-app-db
```

5. Set the user permission to `all` for the user `web-app-user` to be able to read and write in the database.

```shell
scw rdb privilege set instance-id=c39e632d-c6c9-4cf4-adcc-0827143f9e1f
database-name=web-app-db user-name=web-app-user permission=all
```

6. Record the database's private IP address (e.g., `172.16.8.3`) for use in the application configuration.

Downloading the Kubeconfig file

Retrieve the kubeconfig file to manage the cluster with `kubectl`.

```shell
scw k8s kubeconfig get cluster-id=248ed689-5d57-418b-8208-179f87d251b0 >
~/.kube/config
```

This command downloads the kubeconfig file and sets it up for `kubectl` to interact with the cluster.

Setting up Database Credentials in Kubernetes

1. Create the `web-app` namespace to ensure it exists for the Secret and Deployment.

```shell
kubectl create namespace web-app
```

2. Create a Kubernetes Secret to store the database credentials (`DB_USER`, `DB_PASSWORD`, `DB_NAME`) using `kubectl create secret generic`.

```shell
echo -n 'web-app-user' > ./db-user
```

```
echo -n 'secret_Pa$$word_99' > ./db-password
echo -n 'web-app-db' > ./db-name
kubectl create secret generic nodejs-backend-secrets -n web-app
--from-file=db-user=./db-user --from-file=db-password=./db-password
--from-file=db-name=./db-name
```

3. Verify the Secret was created successfully:

```
Shell
kubectl get secret nodejs-backend-secrets -n web-app
```

You should see an output similar to:

```
None
NAME                    TYPE     DATA   AGE
nodejs-backend-secrets Opaque   3      10s
```

Inspect the Secret's contents:

```
Shell
kubectl describe secret nodejs-backend-secrets -n web-app
```

Expected output:

```
None
Name:         nodejs-backend-secrets
Namespace:    web-app
Type:         Opaque
Data
====
db-user:      11 bytes
```

```
db-password:  15 bytes
db-name:      10 bytes
```

4. Clean up the temporary files to avoid leaving sensitive data on the disk:

```shell
rm ./db-user ./db-password ./db-name
```

Important: The Secret must exist before applying the Deployment in the next section. If you encounter the `error secret "nodejs-backend-secrets" not found`, re-run the above commands to recreate the Secret and verify it exists before proceeding.

Creating the Node.js Backend Application

1. Create a directory for the Node.js application:

```shell
mkdir nodejs-backend-app
cd nodejs-backend-app
```

2. Create a file named `package.json` to define dependencies:

```json
{
  "name": "nodejs-backend",
  "version": "1.0.0",
  "description": "Simple Node.js backend for PostgreSQL",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
```

```
    "express": "^4.17.1",
    "pg": "^8.7.3"
  }
}
```

3. Create a file named `server.js` with a simple Express application that connects to the PostgreSQL database, creates a table, and provides endpoints to write and read data:

JavaScript

```
const express = require('express');
const { Pool } = require('pg');

const app = express();
const port = 3000;

// Database configuration from environment variables
const pool = new Pool({
  host: process.env.DB_HOST ? process.env.DB_HOST.split(':')[0] : 'localhost', //
Fallback to localhost
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  port: process.env.DB_HOST && process.env.DB_HOST.includes(':') ?
parseInt(process.env.DB_HOST.split(':')[1]) : 5432
});

// Middleware to parse JSON
app.use(express.json());

// Create a table if it doesn't exist
async function initDb() {
  try {
    await pool.query(`
      CREATE TABLE IF NOT EXISTS messages (
        id SERIAL PRIMARY KEY,
        content TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
      )
    `);
    console.log('Table "messages" is ready');
  } catch (err) {
    console.error('Error creating table:', err);
    // Do not exit, allow server to keep running
  }
}

// Endpoint to write a message to the database
app.post('/messages', async (req, res) => {
  const { content } = req.body;
  if (!content) {
    return res.status(400).json({ error: 'Content is required' });
  }
  try {
    const result = await pool.query(
      'INSERT INTO messages (content) VALUES ($1) RETURNING *',
      [content]
    );
    res.json(result.rows[0]);
  } catch (err) {
    console.error('Error inserting message:', err);
    res.status(500).json({ error: 'Database error', details: err.message });
  }
});

// Endpoint to read all messages from the database
app.get('/messages', async (req, res) => {
  try {
    const result = await pool.query('SELECT * FROM messages ORDER BY created_at
DESC');
    res.json(result.rows);
  } catch (err) {
    console.error('Error fetching messages:', err);
    res.status(500).json({ error: 'Database error', details: err.message });
  }
});

// Health check endpoint
```

```
app.get('/health', async (req, res) => {
  try {
    await pool.query('SELECT 1'); // Test database connection
    res.json({ status: 'ok', database: 'connected' });
  } catch (err) {
    res.json({ status: 'ok', database: 'disconnected', error: err.message });
  }
});

// Initialize database and start server
initDb().then(() => {
  app.listen(port, () => {
    console.log(`Server running on port ${port}`);
  });
}).catch(err => {
  console.error('Failed to initialize database:', err);
  // Keep server running for debugging
  app.listen(port, () => {
    console.log(`Server running on port ${port} despite database error`);
  });
});
```

4. Create a `Dockerfile` to build the Node.js application image:

```
None
FROM node:16
WORKDIR /app
COPY package.json .
RUN npm install
COPY server.js .
EXPOSE 3000
CMD ["npm", "start"]
```

5. Build and push the Docker image to Scaleway Container Registry (replace `web-app-registry` and `pl-waw` with your registry namespace and region):

```shell
scw registry login
docker build --platform linux/amd64 -t
rg.pl-waw.scw.cloud/web-app-registry/nodejs-backend:latest .
docker push rg.pl-waw.scw.cloud/web-app-registry/nodejs-backend:latest
```

Deploying a web application

Deploy the Node.js backend and Nginx frontend, ensuring they communicate with the database over the Private Network. The Node.js backend uses the database credentials from the Kubernetes Secret, and Nginx forwards requests to the backend.

Deploying the `backend` (Node.js API)

1. Create a `backend-deployment.yaml` file to deploy the Node.js API, using the custom Docker image.

```none
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-backend
  namespace: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodejs-backend
  template:
    metadata:
      labels:
        app: nodejs-backend
    spec:
      containers:
        - name: nodejs-backend
          image: rg.pl-waw.scw.cloud/web-app-registry/nodejs-backend:latest
          env:
            - name: DB_HOST
```

```
              value: "172.16.8.3" # Replace with your database's private IP
            - name: DB_USER
              valueFrom:
                secretKeyRef:
                  name: nodejs-backend-secrets
                  key: db-user
            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: nodejs-backend-secrets
                  key: db-password
            - name: DB_NAME
              valueFrom:
                secretKeyRef:
                  name: nodejs-backend-secrets
                  key: db-name
          ports:
            - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: nodejs-backend-service
  namespace: web-app
spec:
  selector:
    app: nodejs-backend
  ports:
    - protocol: TCP
      port: 80        # external port
      targetPort: 3000 # matches containerPort
  type: LoadBalancer
```

Note: Replace `web-app-registry` in the image name with your Container Registry namespace. Ensure the `nodejs-backend-secrets` Secret exists in the `web-app` namespace before applying this YAML. Run `kubectl get secret nodejs-backend-secrets -n web-app` to verify. If the Secret is missing, recreate it as described in the "Setting up Database Credentials in Kubernetes" section. Replace `172.16.8.3` with the actual database private IP if different.

2. Apply the deployment:

```
Shell
kubectl apply -f backend-deployment.yaml
```

The deployment runs 3 replicas of the Node.js backend, distributed across the node pools in different AZs. The `DB_HOST` environment variable uses the database's private IP for secure communication over the Private Network. The `DB_USER`, `DB_PASSWORD`, and `DB_NAME` are sourced from the Kubernetes Secret (`nodejs-backend-secrets`).

Verifying the deployment

1. Check that pods are running and distributed across AZs:

```
Shell
kubectl get pods -n web-app -o wide
```

You should see an output similar to the following example:

```
None
NAME                                READY   STATUS    RESTARTS   AGE   IP
NODE
nodejs-backend-abc123-xyz           1/1     Running   0          5m
100.64.6.118  scw-pool-pl-waw-1-...
nodejs-backend-abc123-def           1/1     Running   0          5m
100.64.2.30   scw-pool-pl-waw-2-...
nodejs-backend-abc123-ghi           1/1     Running   0          5m
100.64.4.74   scw-pool-pl-waw-3-...
```

2. Use `kubectl get svc` to list services in the `web-app` namespace:

```shell
kubectl get svc nodejs-backend-service -n web-app
```

3. Get the `LoadBalancer` external IP:

```shell
NAME                    TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
AGE
nodejs-backend-service  LoadBalancer   10.69.123.44    51.159.x.x      80/TCP
5m
```

Take note of the EXTERNAL-IP (e.g., 51.159.x.x).

4. Then, in a new terminal, test the endpoints:

- Health check:

```shell
curl http://51.159.x.x/health
```

Expected output:

```none
{"status":"ok"}
```

- Write a message:

```shell
curl -X POST http://51.159.x.x/messages \
       -H "Content-Type: application/json" \
       -d '{"content":"Hello, PostgreSQL!"}'
```

Expected output (example):

```
None
{"id":1,"content":"Hello, PostgreSQL!","created_at":"2025-07-17T11:00:00.123Z"}
```

- Read messages:

```
Shell
curl http://51.159.x.x/messages
```

Expected output (example):

```
None
[{"id":1,"content":"Hello, PostgreSQL!","created_at":"2025-07-17T11:00:00.123Z"}]
```

Monitoring and scaling

- **Monitor cluster health**: The autoheal feature automatically restarts or replaces unresponsive nodes after 15 minutes.
- **Scale node pools**: Autoscaling adjusts node counts within the `min-size` and `max-size` limits based on workload. To manually scale a pool:

```
Shell
scw k8s pool update pool-id=b7e85aa1-a319-4cc5-a47a-0bef30d859c7 size=3
```

Cleaning up (optional)

Once done, delete the cluster and its additional resources:

```shell
scw k8s cluster delete 248ed689-5d57-418b-8208-179f87d251b0
with-additional-resources=true
```

Warning: This permanently deletes the cluster, pools, and associated resources like Load Balancers.

Learning summary

After completing this tutorial, you will have learned how to:

- **Secure your cluster**: Isolate communication with a **Private Network** and store credentials with **Kubernetes Secrets**.
- **Deploy a real application:** Run a Node.js API that interacts with a managed PostgreSQL database.
- **Ensure resilience**: Use multi-zone node pools for high availability.
- **Scale effectively**: Enable **autoscaling** to match resources to demand.

- **Expose services cleanly**: Use a **Kubernetes LoadBalancer** to make your app available on the internet without needing a custom reverse proxy.

With these skills, you'll be prepared to go beyond theory and deploy applications in Kubernetes clusters that are closer to real-world setups.

Next steps (optional challenges):

- Add an Ingress controller with TLS for advanced traffic management.
- Deploy a frontend service alongside the backend.
- Set up monitoring and logging to observe your cluster in action.