Activity2 — Serverless Framework v4 Core Concepts 🧠

Objective🎯

- Explain what the Serverless Framework v4 does and why we use it.
- Describe the main **keywords** used in `serverless.yml` (service, function, event, provider, resources, stages, variables, outputs, deployment bucket, artifacts) and when to use them.
- Run and interpret core Serverless **commands** (`deploy`, `package`, `invoke`, `invoke local`, `logs`, `info`, `remove`).
- Use essential plugins (install + configure) and run basic local development flows.
- Inspect the CloudFormation template that Serverless generates and know where artifacts are stored.

Prerequisites (one-time per activity)

You must have completed Activity1. Before starting this activity, run:

```shell
aws configure --profile serverless-lab    # set credentials for profile 'serverless-lab'
serverless login                          # link CLI with your Serverless.org account (dashboard)
```

> Note: these two steps are required **once per activity** so Serverless can use your AWS profile and (optionally) the Serverless Dashboard.

Lab setup (quick)

Create a small demo workspace we'll use for hands-on snippets:

```shell
mkdir -p ~/serverless_lab/Activity2_demo
cd ~/serverless_lab/Activity2_demo
```

You can remove this folder later.

---

1) Introduction — what Serverless Framework is

Serverless (idea): a way to build apps where you do not manage servers — AWS runs the compute (e.g., Lambda), you pay per-use.

Serverless Framework (tool): a command-line tool that helps you describe serverless apps in one file (`serverless.yml`) and then deploy them as AWS (CloudFormation) stacks. Think of it like: *write a short recipe → the tool produces the long CloudFormation recipe and tells AWS to build everything for you*.

Use cases: small REST APIs, cron jobs, event-driven pipelines (S3 → Lambda), backends for single page apps, simple data processing.

---

2) Supported providers

Serverless Framework has plugins for multiple clouds (AWS, Azure, GCP, Cloudflare Workers, etc.). In this lab we focus on **AWS**.

---

3) Why use Serverless Framework

- You write a small config file instead of huge CloudFormation templates -> basically Infrastructure as Code (**IaC**).

- Framework auto-creates IAM roles and wiring for you (but you can override them).
- Easily manage multiple stages (dev/prod) with one repo.
- Big plugin ecosystem for packaging, local testing, canary releases.
- Works well with CI/CD pipelines.

---

4) Runtimes

Serverless supports many runtimes (Python, Node, Java, Go). We pick **python3.10** for all function code in this lab. That means your Lambda handlers are `.py` files and the runtime field in `serverless.yml` will be `python3.10`.

---

5) AWS services you can define with Serverless

- **Lambda** — the function code (compute). Use for request handlers or event processors.
- **API Gateway** — HTTP endpoints fronting Lambdas. Use for REST/HTTP APIs.
- **DynamoDB** — NoSQL table for storing items (session, user data).
- **S3** — object storage; trigger a Lambda on an upload.
- **SQS** — queues for decoupling and async processing.
- **SNS** — pub/sub fanout.
- **CloudWatch** — logs, metrics, alarms.
- **Step Functions** — orchestrate long-running workflows. Serverless expresses all of these either via `functions` (with `events`) or via `resources`.

---

6) Core keywords in `serverless.yml` — definition + example + use case

Below each keyword is explained in simple terms and followed by a minimal YAML snippet.

`service`

Meaning: the name of your whole project / unit of deploy.
Use: organizes and names the CloudFormation stack.

```
None
service: todo-service
```

Use case: pick `service` as a short, unique name that describes this application.

`provider`

Meaning: global settings for cloud provider (AWS), runtime, region, account details.
Use: set defaults used by all functions (runtime, region, IAM role behavior).

```
None
provider:
  name: aws
  runtime: python3.10
  region: ap-south-1
  profile: serverless-lab
```

Important fields:

- `name` — cloud provider (aws).
- `runtime` — default runtime for functions.
- `region` — AWS region for deployment.
- `profile` — AWS CLI profile to use.

`functions`

Meaning: all Lambda functions; each entry defines the **handler**, **events**, and optional overrides.
Use: declare function code and triggers.

```
None
functions:
  createTodo:
    handler: handlers.api.create_todo   # file handlers/api.py function
create_todo
    events:
      - httpApi:
          path: /todos
          method: post
```

Use case: define REST endpoints, S3 triggers, SQS listeners etc. Each `function` maps to one Lambda in AWS.

`handler` (within a function)

Meaning: the entry point to your function; format `file.func` (Python).
Use case: `handlers.api.create_todo` means open `handlers/api.py` and call `def create_todo(event, context)`.

`events`

Meaning: the triggers that cause a function to run (HTTP request, S3 event).
Example events (brief):

- `httpApi` / `http` → API Gateway routes.
- `s3` → bucket event (upload).
- `sqs` → queue message.
- `sns` → topic message.
- `schedule` → cron-like scheduled events.

Example:

```
None
events:
  - s3:
      bucket: my-upload-bucket
      event: s3:ObjectCreated:*
```

`resources`

Meaning: direct CloudFormation block; create anything CFN supports (DynamoDB table, S3 bucket).
Use: when you need extra infra that Serverless does not add automatically.

```
None
resources:
  Resources:
    TodoTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: TodoTable
        AttributeDefinitions: ...
```

Note:
In AWS, `CFN` is an abbreviation for AWS CloudFormation, a service that allows you to model and provision AWS resources by defining them in a template written in YAML or JSON.
`resources` are raw CloudFormation—be comfortable reading CFN syntax.

`stages`

Meaning: named deployment environments (e.g., `dev`, `prod`).
Use: you deploy the same service under different stages to isolate them.
Command usage:

```
None
service: my-service

provider:
  name: aws
  runtime: python3.10
  stage: ${opt:stage, 'dev'}   # default = dev if no stage passed
  region: ap-south-1

functions:
  hello:
    handler: handler.hello
    environment:
      TABLE_NAME: ${self:service}-${self:provider.stage}-table
```

How it works

- If you run:

```
Shell
sls deploy --stage dev
```

→ environment variable `TABLE_NAME = my-service-dev-table`

- If you run:

```
Shell
sls deploy --stage prod
```

→ environment variable `TABLE_NAME = my-service-prod-table`

This way, **different stages** get **separate resources** (good isolation between dev and prod).

`variables`

Meaning: placeholders and lookups used in `serverless.yml` (self, env, ssm, file).
Examples & use-cases:

- `${self:provider.region}` → refer to a value in the same file.
- `${env:MY_VAR}` → read an environment variable from your shell.
- `${ssm:/path/to/secret~true}` → fetch secure parameter from SSM.
- `${file(./config.${opt:stage}.yml):dbTable}` → include values from external file.

Why use: keep secrets out of repo, vary config by stage, reuse values.

`outputs`

Meaning: Values that CloudFormation will export or display once the stack is deployed — such as **API endpoints** or **ARNs** of created resources.

Use:

- Shown when you run `sls info`.
- Useful for referencing resources across stacks.

Note: An **Amazon Resource Name (ARN)** is a globally unique identifier for an AWS resource (e.g., `arn:aws:lambda:ap-south-1:123456789012:function:my-service-dev-hello`).

Sample YAML (`serverless.yml`)

```
None
service: my-service

provider:
  name: aws
  runtime: python3.10
  region: ap-south-1
```

```
functions:
  hello:
    handler: handler.hello
    events:
      - httpApi:
          path: /hello
          method: get

outputs:
  ApiEndpoint:
    Description: "Base API endpoint"
    Value: !Sub "https://${HttpApi}.execute-api.${AWS::Region}.amazonaws.com"
```

Example command

Shell

```
sls info --stage dev
```

Sample output

Shell

```
Service Information
service: my-service
stage: dev
region: ap-south-1
stack: my-service-dev
resources: 10
api keys:
  None
endpoints:
  GET - https://abc123.execute-api.ap-south-1.amazonaws.com/hello
functions:
  hello: my-service-dev-hello
outputs:
  ApiEndpoint: https://abc123.execute-api.ap-south-1.amazonaws.com
```

This shows the **endpoint URL** under `outputs`, along with function names and other stack info.

`deploymentBucket`

Meaning: The **deployment bucket** is an **S3 bucket** that the Serverless Framework uses internally during deployment.

- Every time you run `sls deploy`, Serverless packages your Lambda code (and dependencies) into a **zip file (artifact)**.
- That zip is uploaded to this **S3 bucket**.
- CloudFormation then uses that artifact from S3 to actually create/update the Lambda functions.

Without this bucket, AWS would not know where to fetch your function code from.

Why S3 is used by Serverless:

1. **Staging area for artifacts** → Lambda requires code to be in S3 before CloudFormation can deploy it.
2. **Version history** → Each deploy uploads a new zip file. You can roll back or inspect old versions.
3. **Scalability** → S3 is highly available, so artifacts are reliably stored during deployment.
4. **Organization** → You can control bucket name, region, encryption, lifecycle rules.
    - Example: auto-expire old artifacts after 7 days to save cost.
    - Example: enforce that artifacts are always in the same region as your Lambdas.

Sample YAML (`serverless.yml`)

```
None
provider:
  name: aws
  runtime: python3.10
  region: ap-south-1
  deploymentBucket:
    name: my-serverless-deploy-bucket
    blockPublicAccess: true       # recommended: no public access
    serverSideEncryption: AES256  # encrypt artifacts at rest
```

```
    maxPreviousDeploymentArtifacts: 5  # keep only last 5 versions
```

Hands-on check:

1. Deploy a service:

```
Shell
sls deploy --stage dev
```

2. Go to **AWS Console → S3 → my-serverless-deploy-bucket**.
3. You'll see a folder structure like:

```
None
serverless/my-service/dev/1664630509834-2022-10-01T12:15:09/my-service.zip
```

- Each folder represents a deployment timestamp.
- Inside: the zipped Lambda artifact uploaded by Serverless.

In short: the **deploymentBucket is the bridge** between your local machine and AWS CloudFormation. It makes sure AWS has the packaged code to deploy.

`artifacts`

Meaning: the deployment bundles — zip files or container images that are uploaded.
Where to look: `.serverless/` folder after `sls package`.
Use: CI pipelines publish these artifacts and feed them to CloudFormation.

---

7) Serverless Framework commands — detailed with hands-on examples

Below are the common commands you will use, what each does, and short hands-on steps.

For the examples assume current dir: `~/serverless_lab/Activity2_demo`

`sls`

Meaning: `sls` is the short form alias for `serverless`.
What it does: scaffolds a small Python service in `demo/`.
Hands-on:

```shell
sls
# Choose template as: AWS Python Simple function
# name project as: demo

cd demo
ls
```

Check: you should see a `serverless.yml` and a handler file. Open `serverless.yml` to inspect.

`app`

Meaning: The `app` keyword in `serverless.yml` is used to link your service to the **Serverless Dashboard** (on serverless.com).
When you specify an `app` name, your deployments, metrics, and secrets can be managed in the Dashboard.
Why use:

- Centralized monitoring (function invocations, errors).
- Team collaboration (share deployments, CI/CD hooks).
- Secure secrets management.
- Easy rollback and deployment history.

Sample YAML (`serverless.yml`)

```none
service: my-service
```

```
app: serverless-lab-app        # app name in Serverless Dashboard
org: my-org-name            # your org in Serverless Dashboard

provider:
  name: aws
  runtime: python3.10
  region: ap-south-1
```

Hands-on:

1. Login to Serverless Dashboard:

```
Shell
serverless login
```

2. After deployment, open https://app.serverless.com → select your `org` → `app` → you'll see:
    - deployed service,
    - stage,
    - endpoints,
    - logs & metrics.

Use `app` and `org` when you want **visibility and collaboration** through the Serverless Dashboard. For solo labs, it's optional, but for real-world projects, it's strongly recommended.

`sls package`

What it does: packages your code and dependencies but **does not** deploy. It writes artifacts to `.serverless/`.

Hands-on:

```shell
sls package
ls -la .serverless
# you will see a zipped artifact and cloudformation template
```

Why useful: debug packaging problems locally; CI often runs `sls package` first. You can find the final AWS Cloudformation json there.

`sls deploy`

What it does: packages, uploads artifacts to S3, generates a CloudFormation template, and runs CloudFormation to create/update resources.
Hands-on (small demo):

```shell
sls deploy --stage dev --region ap-south-1 --aws-profile serverless-lab
# watch the output: it lists created resources and endpoints
```

After deploy: note the API endpoint printed and the CloudFormation stack name in AWS Console.

Common flags: `--stage`, `--region`, `--aws-profile` (or `--profile serverless-lab`).

`sls info`

What it shows: a summary of deployed stack: endpoints, functions, stack name, outputs.
Hands-on:

```shell
sls info --stage dev
```

Use-case: quickly find your API base URL or function ARNs.

`sls invoke`

What it does: calls a deployed Lambda function with a payload.
Hands-on:

```shell
# sls invoke -f <function-name> --data '{"key":"value"}'

sls invoke -f createTodo --data '{"title":"buy milk"}' --stage dev

# prints function output JSON
```

Use-case: basic smoke test without using API Gateway.

`sls invoke local`

What it does: runs your handler **locally** in your machine's Python runtime. No AWS involved.
Hands-on example:

1. Create a simple handler `handlers/api.py`:

```python
# handlers/api.py
def hello(event, context):
    return {"statusCode": 200, "body": "hello from local"}
```

2. `serverless.yml` function:

```
None
functions:
  hello:
    handler: handlers.api.hello
```

3. Run:

```
Shell
sls invoke local -f hello --data '{}'
# Expected output printed in console
```

Use-case: quick unit-like run, faster iteration.
Caveat: local run won't catch **IAM** or **VPC** issues that only appear in AWS.

`sls logs`

What it does: tails CloudWatch logs for a deployed function. **Hands-on:**

```
Shell
# sls logs -f <function> --tail

sls logs -f createTodo --stage dev --tail

# watch logs in real time while invoking API
```

Use-case: debug runtime errors that only appear when function executes in AWS.

`sls remove`

What it does: removes the CloudFormation stack and deletes created resources.
Hands-on:

```shell
sls remove --stage dev --region ap-south-1
```

Important: always run `sls remove` to avoid lingering costs from resources.

---

8) Plugins — how to install/configure and why

What are plugins?

- **Plugins** are add-ons for the Serverless Framework.
- They extend or modify the default behavior of deployments (packaging, testing, monitoring, release strategies, etc.).
- Think of them like **"extensions"** in VSCode or Chrome — the core tool works without them, but plugins add powerful features.

Are they installed by default?

❌ **No**.

- Serverless Framework comes with a **core set of commands** (deploy, invoke, logs, etc.).
- Any **extra capability** (like Python dependency packaging, local API simulation, canary rollouts) requires installing plugins.
- You install them using `npm` inside your service directory.

How to install a plugin

Run this inside your project folder:

```shell
npm init -y   # creates package.json if not already
npm install --save-dev serverless-python-requirements serverless-offline
```

This will add them to your `devDependencies` in `package.json`.

How to enable a plugin

Add them to the `plugins` section of your `serverless.yml`:

```
None
plugins:
  - serverless-python-requirements
  - serverless-offline
```

Common Plugins

1. `serverless-python-requirements`

Why:

- By default, Serverless does not know how to package Python dependencies.
- This plugin reads your `requirements.txt` and bundles dependencies into the Lambda zip.
- It can use Docker (`dockerizePip: true`) to ensure compatibility with Lambda's Linux runtime.

How to configure:

```
None
plugins:
  - serverless-python-requirements

custom:
  pythonRequirements:
    dockerizePip: true   # builds deps inside Docker (matches AWS runtime)
    zip: true            # compress dependencies
    slim: true           # remove unnecessary files
```

Hands-on:

1. Create `requirements.txt` with a small package, e.g.:

```
None
requests==2.31.0
```

2. Run deploy:

```
Shell
sls deploy
```

3. In `.serverless/` you'll see dependencies zipped along with your function.

4. Inside Lambda console → Code → check that `requests` is included.

2. `serverless-offline`

Why:

- Testing every change by deploying to AWS is slow.
- This plugin lets you run your API Gateway + Lambda **locally** (`http://localhost:3000`).
- Speeds up development and debugging.

How to configure:

```
None
plugins:
  - serverless-offline
```

Hands-on:

```shell
sls offline start
```

- Now you can hit your endpoint locally:

```shell
curl http://localhost:3000/hello
```

- Useful for frontend teams who want a working API without needing AWS access.

3. `serverless-deployment-bucket`

Why:

- By default, Serverless auto-creates a random-named S3 bucket for artifacts.
- This plugin lets you **control the bucket name and settings** (encryption, lifecycle).
- Useful for compliance, cost control, and centralizing artifacts.

How to configure:

```none
plugins:
  - serverless-deployment-bucket

provider:
  deploymentBucket:
    name: my-serverless-deploy-bucket
    blockPublicAccess: true
    serverSideEncryption: AES256
    lifecycleRules:
      - expirationInDays: 7   # delete old artifacts after 7 days
```

Hands-on:

1. Deploy once.

2. Check S3 console → you'll see all artifacts go into `my-serverless-deploy-bucket`.

4. `serverless-plugin-canary-deployments` (advanced)

Why:

- Sometimes deploying a new version of Lambda directly is risky.
- Canary deployment = send **10% traffic** to new version → if no errors, then send 100%.
- Helps reduce risk of breaking production.

How to configure:

```
None
plugins:
  - serverless-plugin-canary-deployments

functions:
  hello:
    handler: handler.hello
    deploymentSettings:
      type: Canary10Percent5Minutes
      alias: Live
```

Hands-on:

1. Deploy function with canary settings.
2. Serverless will automatically configure Lambda Aliases to shift 10% traffic to new version.
3. After 5 minutes (if no rollback), 100% of traffic is shifted.

In short: **Plugins are not auto-installed**. You decide which ones you need, install them via `npm`, and enable in `serverless.yml`. Each plugin solves a specific problem — packaging, local testing, deployment control, or safe rollouts.

---

9) Local development vs deployed behavior — pitfalls & tips

Local (fast feedback)

- `sls invoke local` executes your code locally with your machine's environment.
- `sls offline` simulates HTTP endpoints.

Deployed (real)

- AWS imposes IAM, network (VPC), cold-starts, and limits that local can't simulate.
- Some issues only appear after deploy: missing IAM permissions, incorrect environment variables.

Tips

- Use `dockerizePip: true` in `serverless-python-requirements` when you depend on compiled libraries (ensures manylinux wheels).
- Use local DynamoDB (`amazon/dynamodb-local`) when you want a local copy of the DB (**docker** run provided below).
- Test both local and deployed — local for fast iteration, deployed for final verification.

DynamoDB local quick start

```shell
docker run -p 8000:8000 amazon/dynamodb-local
# in another shell:
aws dynamodb list-tables --endpoint-url http://localhost:8000
```

---

10) Internal process — what happens during `sls deploy` (step-by-step)

1. **Parse `serverless.yml`** and resolve variables.
2. **Build packages**: create zipped artifacts (or container images) per function or per service.
3. **Upload** artifacts to the deployment S3 bucket.
4. **Generate CloudFormation template** (written to `.serverless/cloudformation-template-update-stack.json`).
5. **Call CloudFormation** to create/update the stack. CloudFormation creates all AWS resources declared.
6. **Return outputs** and print endpoints.

How to inspect generated template

```
Shell
sls package
# then:
less .serverless/cloudformation-template-update-stack.json
```

Reading that file shows you the exact resources CloudFormation will create — helpful for debugging and for understanding what Serverless translates to.

---

12) Variables in `serverless.yml` — expanded with concrete use examples

Variables let you keep configuration DRY, pull secrets from secure stores, and change behavior per stage/region without editing the YAML. Below are the variable types we use in this lab, each with (A) how it looks in `serverless.yml` and (B) a short example that **shows the variable being referenced/used**.

`${self:}` — refer to values in the same `serverless.yml`

Definition (declare a value):

```
None
service: todo-service

provider:
  stage: ${opt:stage, 'dev'}   # default stage = dev
```

Usage (where the resolved value is used):

```
None
resources:
  Resources:
    TodoTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${self:custom.tablePrefix}-table   # e.g.,
todo-service-dev-table
```

What happens: if you run `sls deploy --stage prod`, `${self:custom.tablePrefix}` becomes `todo-service-prod`.

`${env:VAR}` — read a shell environment variable

Definition (use an env var in YAML):

```
None
provider:
  name: aws
  runtime: python3.10

functions:
  create:
    handler: handlers.create.handler
    environment:
      TODOS_TABLE: ${env:TODOS_TABLE}   # pulled from your shell env
```

Usage (set and run):

```shell
export TODOS_TABLE="my-todos-dev"
sls deploy --stage dev
# In AWS Lambda, environment variable TODOS_TABLE will be "my-todos-dev"
```

Tip: Good for local overrides or CI secrets injected by the pipeline.

`${ssm:/path/to/param~true}` — fetch secure string from SSM Parameter Store

Definition (reference an SSM parameter in YAML):

```
provider:
  name: aws

functions:
  worker:
    handler: handlers.worker.handler
    environment:
      DB_PASSWORD: ${ssm:/myapp/dev/db_password~true}   # ~true = decrypt
SecureString
```

Usage (how to create and then read):

```shell
# create a SecureString in SSM (one-time, in AWS CLI)
aws ssm put-parameter --name /myapp/dev/db_password --value "s3cr3t" --type
SecureString --overwrite

# then deploy; Serverless will fetch and inject DB_PASSWORD at deploy/runtime
sls deploy --stage dev
```

What happens: Serverless resolves the SSM value at deploy-time (or at runtime if configured) and injects it into the function env. Use `~true` to tell Serverless to decrypt the SecureString.

`${file(./config.${opt:stage}.yml):dbTable}` — load from an external file

External file `config.dev.yml`:

```
None
dbTable: todo_table_dev
apiPrefix: /dev
```

`serverless.yml` using the file (definition + usage):

```
None
provider:
  stage: ${opt:stage, 'dev'}

custom:
  config: ${file(./config.${opt:stage}.yml)}   # loads config.dev.yml for dev

functions:
  list:
    handler: handlers.list.handler
    environment:
      TODOS_TABLE: ${self:custom.config.dbTable}   # becomes todo_table_dev
    events:
      - httpApi:
          path: ${self:custom.config.apiPrefix}/todos   # /dev/todos
          method: get
```

Usage (switch stage):

```
Shell
sls deploy --stage dev    # loads config.dev.yml
sls deploy --stage prod   # would load config.prod.yml if present
```

Why: lets you maintain stage-specific settings in small files (DB names, endpoints, limits).

Quick summary / rules of thumb

- Use `${self:}` for reusing values declared in the same YAML.
- Use `${env:}` for per-machine or CI-provided values (secrets can be injected by pipelines).
- Use `${ssm:...~true}` for secrets stored in AWS SSM SecureString (requires IAM permission to read).
- Use `${file(...)}` to keep large or stage-specific configs outside the main `serverless.yml`.

---

12) Where to look for more info and how to debug

- Generated CloudFormation: `.serverless/cloudformation-template-update-stack.json` — shows what will be applied.
- CloudFormation Console → **change sets** / **events** — useful when deploy fails (errors reported here).
- **CloudWatch** Logs for function traces.
- `sls logs -f <fn>` and AWS Console for detailed debugging.
- `sls package` + inspect artifact to ensure all files are included.

---

13) Notes, best practices & small cautions (simple language)

- **Never** commit `~/.aws/credentials` or secrets into Git. Use environment variables or SSM/Secrets Manager.
- Use `serverless remove` when done to avoid ongoing costs.

- Prefer **least-privilege IAM** for deploy accounts — don't use AdministratorAccess for final tests.
- Use `serverless-python-requirements` with `dockerizePip: true` when your Python packages include native binaries.
- Local tests are helpful but always verify in AWS because of IAM/VPC differences.
- Keep function package sizes small to reduce cold starts. Consider Layers for shared libraries.

Final note — evaluation

There is no evaluation for Activity2.

This activity is purely for learning so you are prepared for Activity3 (hand-held project).

Make sure you can run and explain the commands above — that knowledge is required for later hands-on work. Do attempt the quiz.

═══ END OF DOCUMENT ═══