iac-dynamic-web

Terraform project to deploy a serverless dynamic web endpoint on AWS.

Renato Boegeholz <renatoboegeholz@gmail.com>

Table of Contents

1. Solution Overview

The solution provisions a serverless application on AWS using Terraform.
It consists of:

- An **SSM Parameter** that stores a dynamic string.
- An **AWS Lambda function** that retrieves the parameter and returns an HTML page.
- An **API Gateway (HTTP API)** that exposes the Lambda via a public URL.
- Supporting **IAM roles and permissions** to allow secure interactions.

The HTML page served is:

```
<h1>The saved string is {dynamic string}</h1>
```

The dynamic string is read at runtime from the SSM Parameter Store. This allows updates via the AWS CLI or Console without requiring a redeployment of infrastructure or code.
Any user accessing the URL will see the latest saved string.

Flow:

1. **Terraform** provisions:
    - SSM Parameter (with default string).
    - Lambda function with environment variable PARAM_NAME.
    - API Gateway that routes requests to Lambda.
2. **Lambda**:
    - On request, Lambda reads the string from the SSM Parameter Store.
    - Builds an HTML page with the string.
3. **API Gateway**:
    - Serves the Lambda response over a stable HTTP endpoint.
4. **User**:
    - Accesses the URL in a browser.
    - Can update the message using the helper script (update_string.sh).

2. Design Decisions

- **Terraform** was chosen for Infrastructure as Code to ensure reproducible, auditable, and easily maintainable deployments, and to simplify cleanup of resources.
- **API Gateway (HTTP API)** was selected over REST API due to lower cost, simpler configuration, and faster deployment, while still fully supporting Lambda integration.
- **IAM Roles with managed policies** were used to grant Lambda least-privilege access, specifically to the SSM Parameter needed, balancing security and simplicity.
- **AWS SSM Parameter Store** was chosen for storing the dynamic string because it provides a centralized, secure, and versioned key-value store, allowing updates without code redeployment.
- The **Lambda function retrieves the SSM parameter** on every request, ensuring the string is always up to date for all users, without the need for cache invalidation or redeployments.
- **Terraform outputs** provide the API URL automatically, simplifying testing, demonstration, and integration into other tools or scripts.

3. Discussion of Available Options

Several approaches were considered for serving the HTML page:

- **Amazon S3 Static Website Hosting**
  - Pros: Extremely low-cost, fast, globally distributed with optional CloudFront integration. Very simple for static content.
  - Cons: Not suitable for truly dynamic content without additional services (e.g., Lambda@Edge or JavaScript fetching values), which adds complexity.

- **EC2 or ECS-based Web Server**
  - Pros: Full control over the server and runtime environment. Can handle dynamic content and complex application logic.
  - Cons: Requires provisioning and maintaining servers, scaling, and patching. Higher operational overhead and cost for a simple dynamic string.

- **API Gateway with Mock Integration**
  - Pros: Can return static or templated responses without Lambda. Minimal cost for very simple use cases.
  - Cons: Limited flexibility for dynamic content; any update to the response requires redeploying the API configuration.

- **AWS Lambda (chosen option)**
  - Pros: Fully serverless, no servers to manage. Integrates seamlessly with API Gateway for HTTP access. Can dynamically read SSM Parameter values at runtime, ensuring up-to-date content.
  - Cons: Adds a small cold-start latency on the first request. Each request triggers a Lambda invocation, which may be slightly more expensive at very high traffic volumes.

Decision: AWS Lambda was selected for its simplicity, fully serverless nature, and runtime flexibility, allowing the web page to always reflect the latest SSM parameter value without redeployment.

Another aspect considered was how to make the string dynamic:

- **Environment Variables in Lambda**
  - Pros: Simple and very fast to implement; no additional AWS services needed.
  - Cons: Updating the value requires redeploying the Lambda, which is not practical for frequent or runtime changes.

- **AWS Secrets Manager**
  - Pros: Highly secure, supports automatic rotation, audit logging, and is ideal for storing sensitive data such as credentials or API keys.
  - Cons: Higher cost, unnecessary for a simple, non-sensitive string; adds extra complexity in setup and permissions.

- **DynamoDB or S3**
  - Pros: Allows storing structured or larger data, and can be easily scaled for multi-user or multi-page scenarios.
  - Cons: Overkill for a single string, increases infrastructure complexity, and incurs extra cost.

- **AWS SSM Parameter Store (chosen option)**
  - Pros: Secure, centralized, versioned, and updatable at runtime. Supports fine-grained IAM policies and free-tier usage. Updating the value requires no redeployment, making it ideal for dynamic content.
  - Cons: Adds one extra AWS API call per request, which has minimal impact for low to moderate traffic.

Decision: AWS SSM Parameter Store was selected for its balance of simplicity, flexibility, security, and cost-effectiveness, making it a very suitable choice for the requirements of this challenge.

4. Scalability Considerations

Although the current solution is designed for simplicity and demonstration purposes, several factors affect how it would scale under higher load or in a production environment:

Lambda Scaling
- AWS Lambda automatically scales to handle concurrent requests, so the solution can serve multiple users simultaneously without manual provisioning.
- Consideration: High concurrency may increase cold starts and API Gateway invocation costs. Using provisioned concurrency or warm-up strategies can mitigate latency spikes.

API Gateway
- HTTP APIs scale automatically and can handle thousands of requests per second.
- Consideration: For extremely high traffic or global distribution, adding CloudFront in front of the API can reduce latency and offload repeated requests.

SSM Parameter Access

- Each Lambda invocation reads the dynamic string from SSM Parameter Store, which works well for moderate traffic.
- Consideration: For very high request rates, repeated API calls to SSM could introduce latency. Potential optimizations include:
  - Caching the parameter inside Lambda (with refresh logic).
  - Using a Lambda layer or environment variable updated periodically by a scheduled process.

Monitoring and Scaling Metrics

- AWS CloudWatch metrics for Lambda and API Gateway provide visibility into:
  - Request counts and concurrency
  - Latency and error rates
- Consideration: Setting alarms and dashboards helps detect bottlenecks early and scale or optimize resources proactively.

5. Future Enhancements

With more time, the solution could be extended by:

Improved Security

- Restrict IAM policy to only the specific SSM parameter used by the Lambda (rather than broad SSM read access).
- Use SecureString in SSM for sensitive values.
- Enable AWS KMS encryption for sensitive parameters.

Performance Optimizations

- Cache the SSM parameter in Lambda (one option could be using environment variables set at deploy time or Lambda extensions with refresh logic).
- Add CloudFront in front of API Gateway for caching, latency reduction, and global delivery.

Developer Experience

- Add end-to-end automated tests for the Lambda (unit + integration).
- Provide an example Makefile or task runner for building and deploying.
- Expand Terraform variables to make stage name, and logging levels configurable.
- Add local testing setup (for example using LocalStack or AWS SAM CLI).

Scalability Features

- Add CloudWatch dashboards and alarms for monitoring API usage, errors, and latency.
- Add request rate limiting (via API Gateway usage plans) for protection against abuse.

Production-Readiness

- Introduce a CI/CD pipeline (using GitHub Actions or AWS CodePipeline) for automated build, test, and deploy.
- Add staging environments (dev, test, prod) with isolated parameter namespaces.
- Define infrastructure compliance checks (by using Terraform Cloud policies or AWS Config).