Bern Hack Night

Mastering Cloud-Native computing at zero cost

Michael Morandi, Architect / Lead Engineer

Bern, June 2024

ti&m

What you'll need

Essentials

- AWS account (not older than one year)
- A web browser
- Git

Convenient

- AWS CLI
- Visual Studio Code
- NodeJs

What you'll need

Essentials

- AWS account (not older than one year)
- A web browser
- Git installed

Convenient

- AWS CLI
- Visual Studio Code
- NodeJs

https://t.ly/l29yC

Clone the git repo with the slides and code snippets

Cloud native application

A cloud-native application is specifically designed to leverage the benefits of cloud computing. These applications are built to be **scalable**, **resilient**, and easily manageable. They are composed of **loosely coupled**, independently deployable microservices, all managed using **containerization** technologies.

Cloud-native applications embrace **serverless computing**, enabling resiliency, rapid development & deployment, and can be scaled to meet demand.

Distilled from the CNCF definition

Hyperscalers offer generous free tiers

Cost-effectiveness: The free tiers allow users to use cloud services up to specified limits for free, which can help them save money on their cloud computing costs.

Experimentation and learning: The free tiers allow you to experiment with different cloud services and technologies without worrying about upfront costs.

Proof of concept: Free tiers allow you to develop and test proof-of-concept applications before you commit to a paid plan. This can help you validate your ideas and ensure that cloud computing fits your needs well.

Scalability: The free tiers can be used to build scalable applications that can grow as your needs change.

Free as in ‘free beer’

Some examples

COMPUTE

Free Tier — ALWAYS FREE

AWS Lambda

1 Million

free requests per month

Compute service that runs your code in response to events and automatically manages the compute resources.

1,000,000 free requests per month

Up to 3.2 million seconds of compute time per month

aws

MOBILE

Free Tier — ALWAYS FREE

Amazon SNS

1 Million

publishes

Fast, flexible, fully managed push messaging service.

1,000,000 Publishes

100,000 HTTP/S Deliveries

1,000 Email Deliveries

aws

DATABASE

Free Tier — ALWAYS FREE

Amazon DynamoDB

25 GB

of storage

Fast and flexible NoSQL database with seamless scalability.

25 GB of Storage

25 provisioned Write Capacity Units (WCU)

25 provisioned Read Capacity Units (RCU)

Enough to handle up to 200M requests per month.

aws

Shared

from $0/month

Try for Free

(i) Free forever for free clusters

For learning and exploring MongoDB in a cloud environment. Basic configuration options.

- ✓ 512MB to 5GB of storage
- ✓ Shared RAM
- ✓ Upgrade to dedicated clusters for full functionality
- ✓ No credit card required to start

mongoDB. Atlas

https://aws.amazon.com/free https://cloud.google.com/free https://azure.microsoft.com/pricing/free-services

Beware of the vendor lock-in

AWS Lambda functions

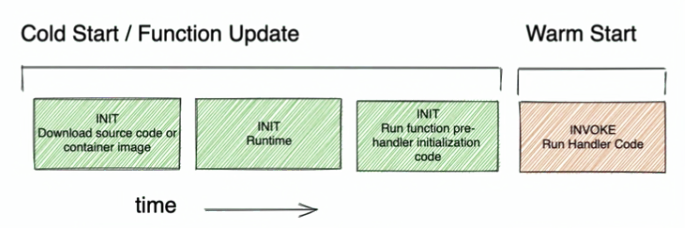
- **Event-driven execution**: Lambda functions are triggered by events, such as changes to data in an S3 bucket or an HTTP request. This makes them ideal for handling asynchronous tasks and building serverless applications.
- **High scalability**: Lambda functions can automatically scale up or down to handle changes in demand. This makes them ideal for handling unpredictable workloads.
- **Flexible**: Lambda functions can be written in a variety of programming languages, including Java, Python, Node.js, and Go.
- **Reduced operational overhead**: Lambda functions eliminate the need to provision and manage servers, resulting in significant time and cost savings.
- **Increased agility**: Lambda functions can help you be more agile by making it easier to experiment with new ideas and features.

```python
import json

def lambda_handler(event, context):
    body = {
        "message": "Hello, World!"
    }

    response = {
        "statusCode": 200,
        "body": json.dumps(body)
    }

    return response
```

Lambda execution environment

Building applications with Lambdas involves thinking in new paradigms

- Lambdas are essentially only run on request (new instance is started on invocation)
- The run in a micro VM called Firecracker with a predefined runtime
- They can be packaged in zip files (shared code in layers) or even built as container images
- Support for frameworks such as Express.js (Node.js), FastAPI (Python), or SpringBoot (Java) exists
- Wide - ranging IDE Support and AWS SAM for local development
- Startup times: cold: < 1s, < 100ms (Node.js and Python runtimes have among the lowest cold start times)



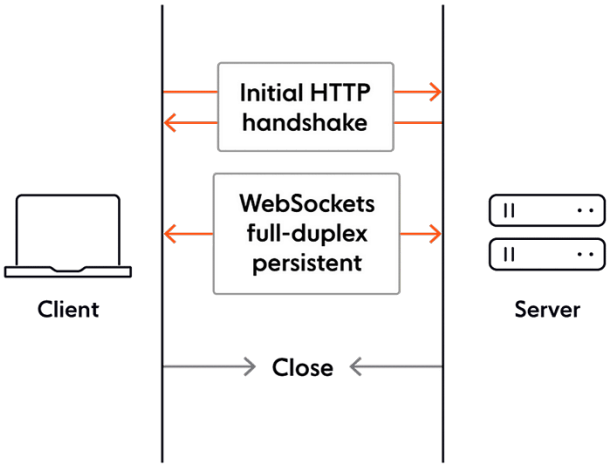
AWS API Gateway

AWS API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.

- **HTTP, REST, and WebSocket APIs:** Supports different types of APIs for various use cases.
- **Integration with AWS Services:** Easily integrates with AWS Lambda, DynamoDB, and more.
- **Monitoring and Metrics:** Provides detailed monitoring and metrics via Amazon CloudWatch.
- **Versioning and Staging:** Supports different stages (development, testing, production) and versioning.
- **Security:** Built-in support for authorization and authentication using IAM, Amazon Cognito, and custom authorizers.
- **Traffic Management:** Offers throttling, request and response transformation, and caching capabilities.
- **Custom Domain Names and SSL:** Supports custom domain names and SSL certificates for API endpoints.
- **SDK Generation:** Automatically generates SDKs for platforms like JavaScript, iOS, and Android to simplify client integration.

WebSockets

The WebSockets protocol enables full-duplex, bidirectional communication over a single TCP connection. It supports real-time interaction between a client and server, ideal for live chats and real-time updates.



Command line tool to interact with WebSockets

```
npm install -g wscat
```

```
wscat -c wss://myapi.aws.amazonaws.com/production
```

WebSocket

Amazon DynamoDB

DynamoDB is a fully managed NoSQL database service provided by AWS that offers fast and predictable performance with seamless scalability.

amazon
DynamoDB

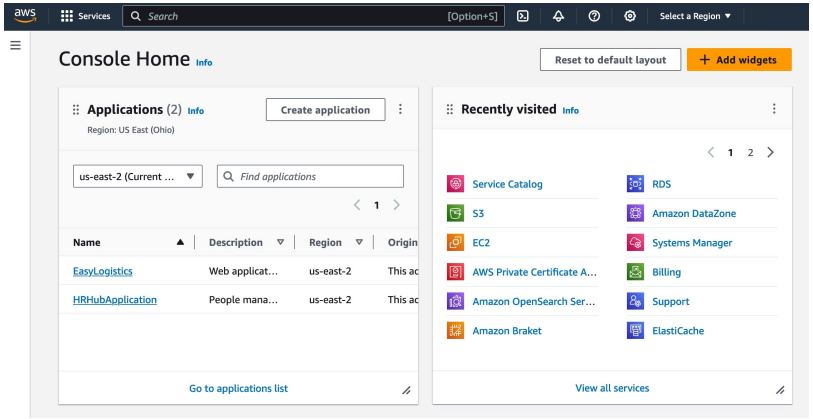
- **Fully Managed**: AWS handles database management tasks like hardware provisioning, setup, configuration, and backups.
- **Flexible Data Model**: Supports key-value and document data models.
- **Global Tables**: Provides multi-region, fully replicated tables for high availability and disaster recovery.
- **High Performance**: Designed for high throughput and low latency at any scale. AWS promises single-digit millisecond response times under optimal conditions.
- **High Availability**: 99.999 when using global tables
- **Scalable**: Automatically scales up or down to handle capacity needs without downtime.
- **Integrated Security**: Offers fine-grained access control with AWS IAM, encryption at rest, and secure communication.
- **Streams**: Captures data changes in real-time, allowing for robust event-driven architectures.

AWS Simple Notification Service (SNS)

AWS SNS is a fully managed messaging service for decoupling and scaling microservices, distributed systems, and serverless applications.

- **Message Delivery:** Push-based, Sends notifications to subscribers using supported protocols (HTTP/S, Email, SMS, SQS, Lambda)
- **Fan-out**: Delivers a single message to multiple endpoints (e.g., send an SMS and an email from a single message).
- **Scalability & Reliability**: Automatic scaling, durable storage across multiple availability zones.
- **Flexible Messaging:** Create and manage topics for message grouping. Use attributes to filter messages to specific subscribers.
- **Security:** Access control via IAM policies, encryption at rest and in transit.
- **Integration:** Integrates with AWS services (Lambda, SQS, CloudWatch) and third-party apps.
- **Cost-Effective:** Only pay for the number of messages published, delivered, and data transfer.

Managing AWS Resources



https://console.aws.amazon.com

aws Cloud Development Kit

AWS CloudFormation

HashiCorp Terraform

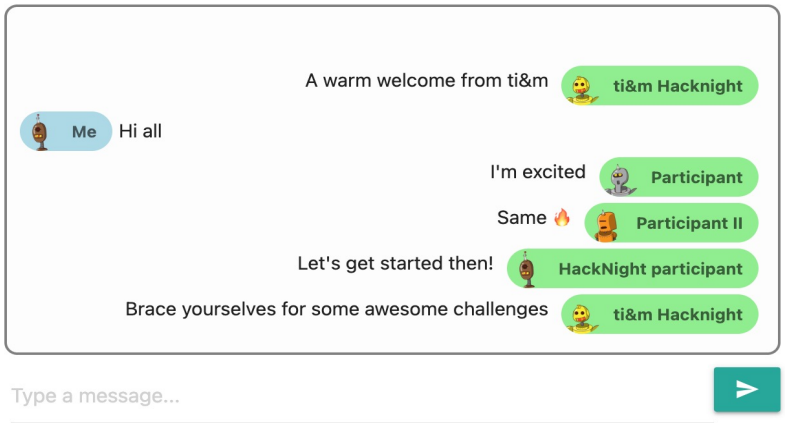
Infastructure as Code (IaC)

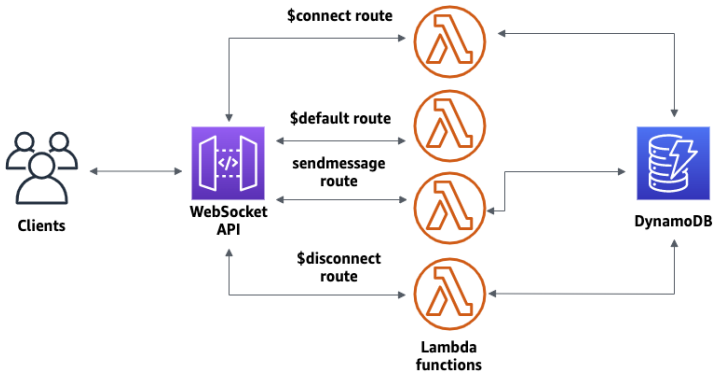
All roads lead to Rome

ti&m

What we’re building…

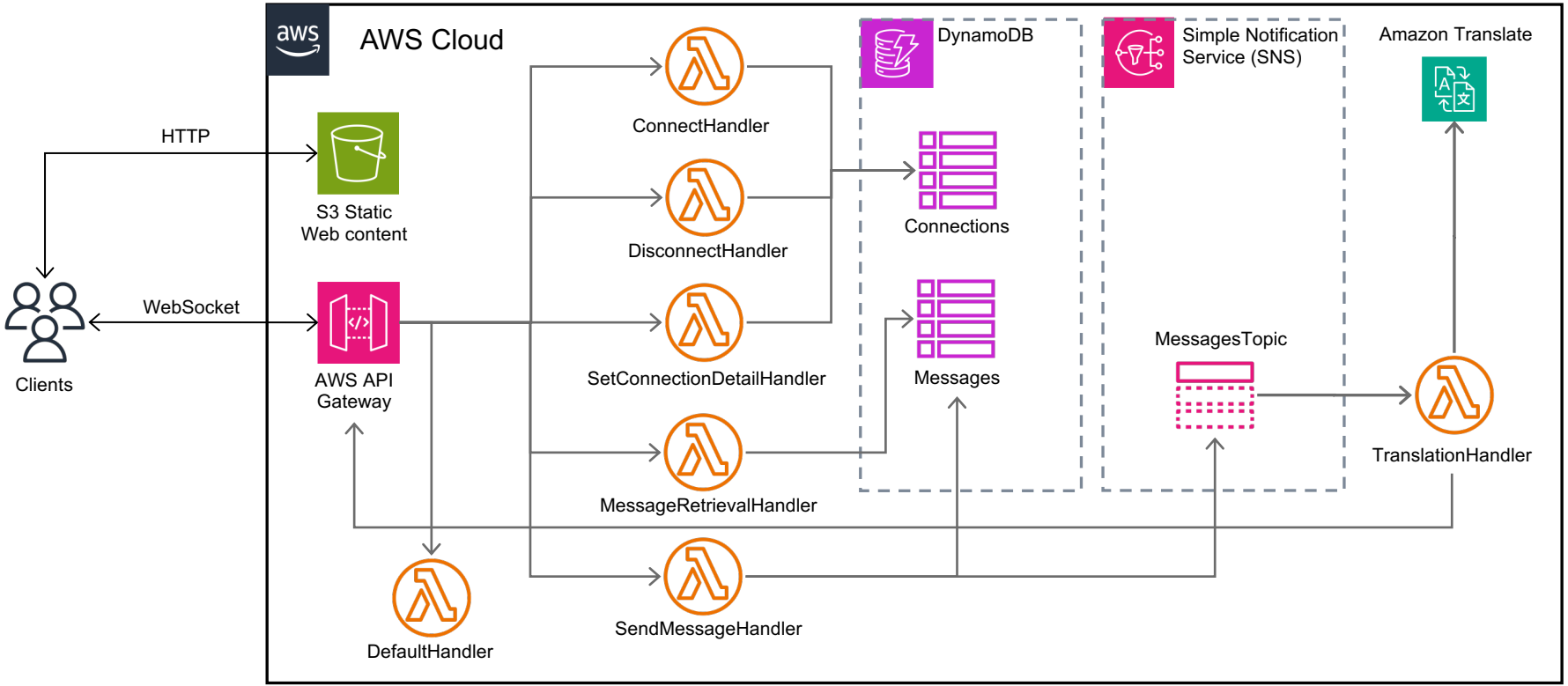
Tonight, we'll create a serverless chat application with a WebSocket API. With a WebSocket API, you can support two-way communication between clients. Clients can receive messages without having to poll for updates.





ti&m

Architecture



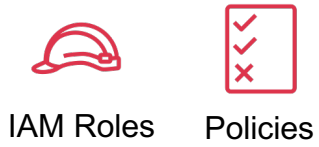
ti&m

Exercise 1
Provision IAM roles and policies

TASK 1

AWS uses a powerful and flexible permission model to control resources. All access is denied by default, so we explicitly have to grant permissions. Our Lambdas require an IAM Role with attached IAM policies to access other resources. For your convenience , we've prepared the necessary roles and permissions.

1. Log into your AWS account and
2. Open the **CloudFormation** service
3. Create a new Stack
    1. From existing template
    2. Upload lambda-roles.yaml from the Git repository
    3. Give the stack a meaningful name like "LambdaRoles"
    4. Enter your AWS account ID without dashes (can be found in the dropdown in the upper right)
    5. Use defaults for the rest
4. Navigate to the IAM service and verify that you see 10 roles



ti&m

Exercise 2

Create the first batch of Lambda functions for API Gateway

TASK 2

API Gateway needs at least three handler functions that are invoked on WebSocket action: One each when a user connects/disconnects and a default handler function (when no route matches)

1. Open **AWS Lambda** in the AWS Console and navigate to the Functions Section
2. Create a new function named *ConnectHandler*
    1. For the runtime, use Node.js 20.x
    2. Change the default execution role to the matching role prepared in Task 1 (e.g. , ConnectHandlerServiceRole)
    3. Once created, copy the source code from the git repo and Deploy it
3. Repeat the above steps for the functions *DisconnectHandler* and *DefaultHandler.*

Exercise 3
Create a Websocket API

1. Open the **API Gateway** service in the AWS console
2. Create a new “WebSocket API”
    1. Choose a name and use `request.body.action` for route selection
    2. Add all predefined routes by clicking their respective buttons
    3. At the integration step connect the Lambdas created in step 2
    4. Give the stage a name, e.g *production*
    5. Create and deploy
3. In the side panel, go to Stages and note the web socket URL (starts with wss://). We’ll use it later.

Exercise 4
Set up DynamoDB tables

TASK 4

Create tables

1. Open the **DynamoDB** console
2. Create a table *Connections*, using *connectionId* (String) as the partition key.
3. Use defaults otherwise

Test the setup

1. Open a terminal (If you don't have node.js installed locally, you can also use **CloudShell** in AWS)
2. Run `wscat –c wss://your-api-url.amazon.com/production`
3. Open the DynamoDB table 'Connections' and click «Explore table items.»
4. You should now see an entry with your IP address!
5. Disconnect, and the entry should now disappear

Exercise 5
Message handler + Frontend

TASK 5

Deploy the message handler

1. Create new **Lambda** functions *SendMessageHandler* and *ConnectionDetailHandler* (like in Exercise 2) and use the appropriate execution roles. Upload the code from the Git Repo and deploy the functions.
2. Then, return to the **API Gateway**, open your WebSocket API, and navigate to the Routes menu. Create new Routes *sendmessage* and *updatedetails* and connect them to the lambdas function created before. Make sure you enable Lambda proxy integration on both functions. Then, deploy the API.

Provide static content (Frontend app)

1. Open the **S3** Service in the AWS console and create a new bucket. Choose a unique name! Also, turn off the public access blocker (uncheck *Block all public access*)
2. In the frontend folder in the git repo, open chatapp.js and insert your wss:// URL from earlier. Then, upload the three files from this folder to the root of the S3 bucket.
3. Enable the Static website hosting in the *Properties* tab and use index.html as the index document.
4. Go to the Permissions tab and add a bucket policy. There's a template in the git repos under aws/S3. Replace "INSERT_BUCKET_NAME" with your actual bucket name
5. The chat app is now operational! To access it, paste the URL found in the Static website hosting section into a browser.

Exercise 6
Messages persistency

TASK 6

Let’s store the messages for users logging in at a later point in time. They should be purged automatically after a specified period of time

1. Open the **DynamoDB** console again.
2. Create a table *Messages* using *userHandle* (String) as the partition key and *timestamp* (Number) as the sort key.
3. Once created, open the *Additional settings* tab and turn on TTL using *timestamp as the attribute key.*
4. Deploy the lambda function *MessageRetrievalHandler*, which you’ll find in the git repo.
5. Add a new route *getmessages* in the **API Gateway**, enable proxy integration, and connect it to the lambda function. Then, deploy the API.
6. Start chatting; you should now see messages appearing in the Messages table.
7. Open a new chat window; you should see past messages when loading.

Bonus exercise

Bonus Exercise 7
Automatic translation

TASK 7

With AWS Translate, we can enhance our chat with simultaneous translation

1. Create a new lambda function *TranslationHandler* from the code in the git repo. Once created, add an environment variable *apiGwEndpoint* with the URL of your API gateway (e.g https://abcdef.execute-api.eu-central-1.amazonaws.com/production)
2. Open the Amazon **Simple Notification Service (SNS)**
3. Create new Standard Topic, name it *MessagesTopic. Make a note of the Topic's ARN.*
4. Now create a subscription with protocol "AWS Lambda" and paste the ARN of the Lambda function created above.
5. Add an environment variable *translationTopic* to the lambda function *SendMessageHandler* with the ARN of your SNS topic
6. Set TRANSLATION_ENABLED in chatapp.js to true and re-upload it to the S3 bucket
7. Test the translation with two chat windows!