Fluently NoSQL

Server Side Swift 2019, Copenhagen

Hello! What an exciting conference, I cannot believe it's almost over!

I'm so excited to share with you the journey to getting an internal application deployed, and the components required to get there.

In particular, we'll talk about integrating Vapor's Fluent ORM with AWS' DynamoDB.

Joe Smith

Site Reliability Engineer at Slack

@Yasumoto

- Performance
- Efficiency
- Reliability
- Scalability

Lots of time spent with dynamic languages

Also a ton of time oncall, and as I'm bouncing between meetings or commutes, rarely have a comfortable way to use my laptop.

Would *love* to be able to use my phone for more incident response!

Agenda

- Patient Road to Production
- DynamoDB
- `AWSSDKSwift`
- Fluent DynamoDB

Launching to Prod

Photo by SpaceX on Unsplash

To build something that your customers rely upon, you need to have utmost confidence in its success.

That's not just something that can happen over a few months.

Need to maintain investment, understanding, and consider what work needs to be done now to enable you in the future.

Swift on the server is *very* new

I love the way the community is working together to build out the necessary frameworks for us to be successful.

Starting with shared, reusable components allows us to build on top of those, solving more specific problems each time.

Photo by Tyler Lastovich on Unsplash

Slack's core Web API is a monolithic application written in Hacklang on HHVM

Also have services in Java & Go for the real-time messaging stack

Tools written in Python, Shell, golang, and Swift

Need to be thoughtful of how & when to introduce a new stack

Photo by Ryan Stone on Unsplash

Start with small, well-scoped tools. For me, that meant finding command-line tools used for things like SSL Certificate management.

I also used to work directly on top of AWS, so it was important to have a reliable AWS client.

So worth your time to Invest in libraries, these will pay off.

As you amass these components, you'll find opportunity to combine technology interests with business use cases

🛑 Emergency Stop

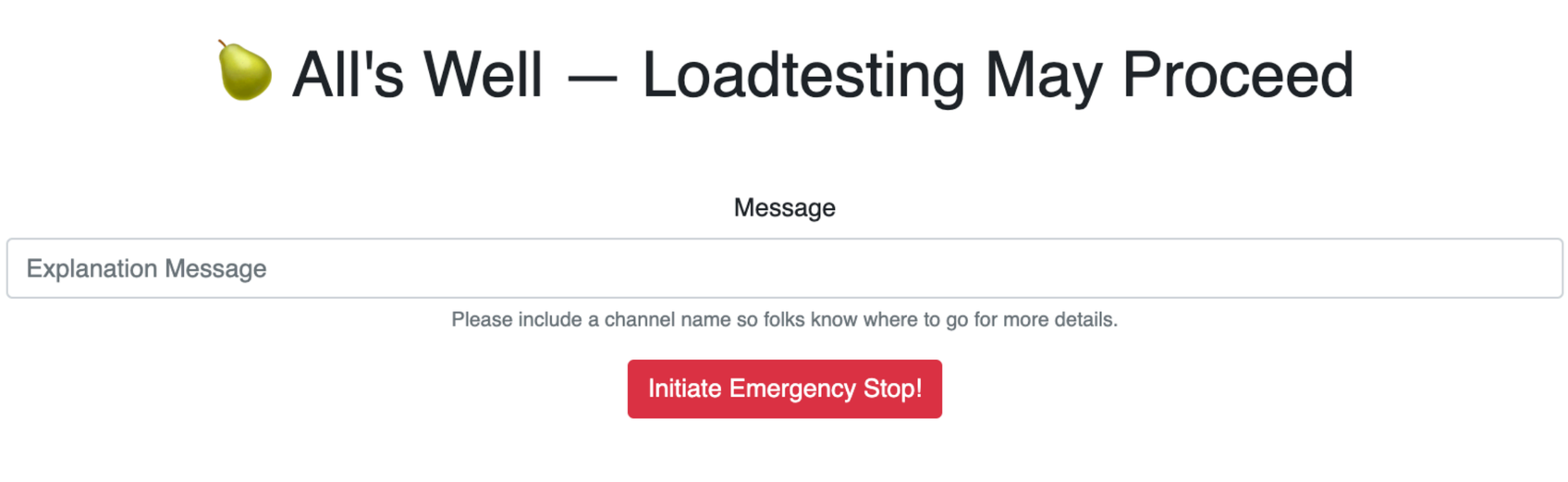
1. What Loadtests are running right now?
2. How do I stop all loadtests NOW?

For me, that opportunity looked like a small service to assist with some major loadtest efforts we were performing.

As a Site Reliability Engineer, we have many tools in our tool belt.

Loadtesting and simulating large-scale events are powerful, but can be dangerous!

We wanted to be able to answer two questions quickly, so we built `Emergency Stop`.



We created a service, and integrated our 3 or 4 loadtesting tools to register themselves with Emergency Stop.

This allowed engineers and incident responders to track what experiments were in-flight

We also wanted a "big red button" to stop anything running immediately.

These tools would also query Emergency Stop to look for whether it was safe to proceed.

Data Structure – `ServiceLock`

1. **ServiceName** 🌍 always `global`
2. **Version** 📜 A historical log
3. **IsIncidentOngoing** 🔴 the "red button"
4. **Username** 🙋‍♀️ Who flipped the switch?
5. **Timestamp** ⌚ When?
6. **Message** ➡️ Where's the incident channel?

Photo by Benjamin Davies on Unsplash

This is our first service written in Swift, leveraging the language, standard library, and the APIs of Fluent & Vapor.

It's running on our internal kubernetes cluster via EKS.

Photo by Dawn Armfield on Unsplash

Also an opportunity to battle-test the AWSSDKSwift

Picking a Datastore (at Slack)

- MySQL
- Something Else

MySQL is the "supported" solution by our Database Reliability Engineering team, but requires maintenance, ongoing support, and an oncall rotation.

We wanted something as low-overhead as possible, able to be controlled by APIs and configuration, not chef and kernel reboots.

Photo by Ian Battaglia on Unsplash

Announced by Amazon CTO Werner Vogels in 2012

Key-Value Storage

Denormalized Data

*Most of our services... store and retrieve data by primary key and do not require the **complex querying** and management functionality offered by an RDBMS.*

-- Dynamo Paper

Reading the Dynamo paper gives some insights into its usage.

The infrastructure engineers realized that for their usecase, most teams were *not* taking advantage of the features offered by a SQL datastore.

If folks could give up some flexibility and take on some constraints, that opened up some interesting options!

Key Concepts

1. Tables
2. Items & Attributes
3. Primary Keys

These are the "nouns" of DynamoDB.

Table

1. Name
2. Hash Key
3. Range Key
4. Capacity

A table is just a collection of data records.

Very similar to a table in MySQL

It has a few properties, the Hash Key & optional Range Key compose the *primary key*

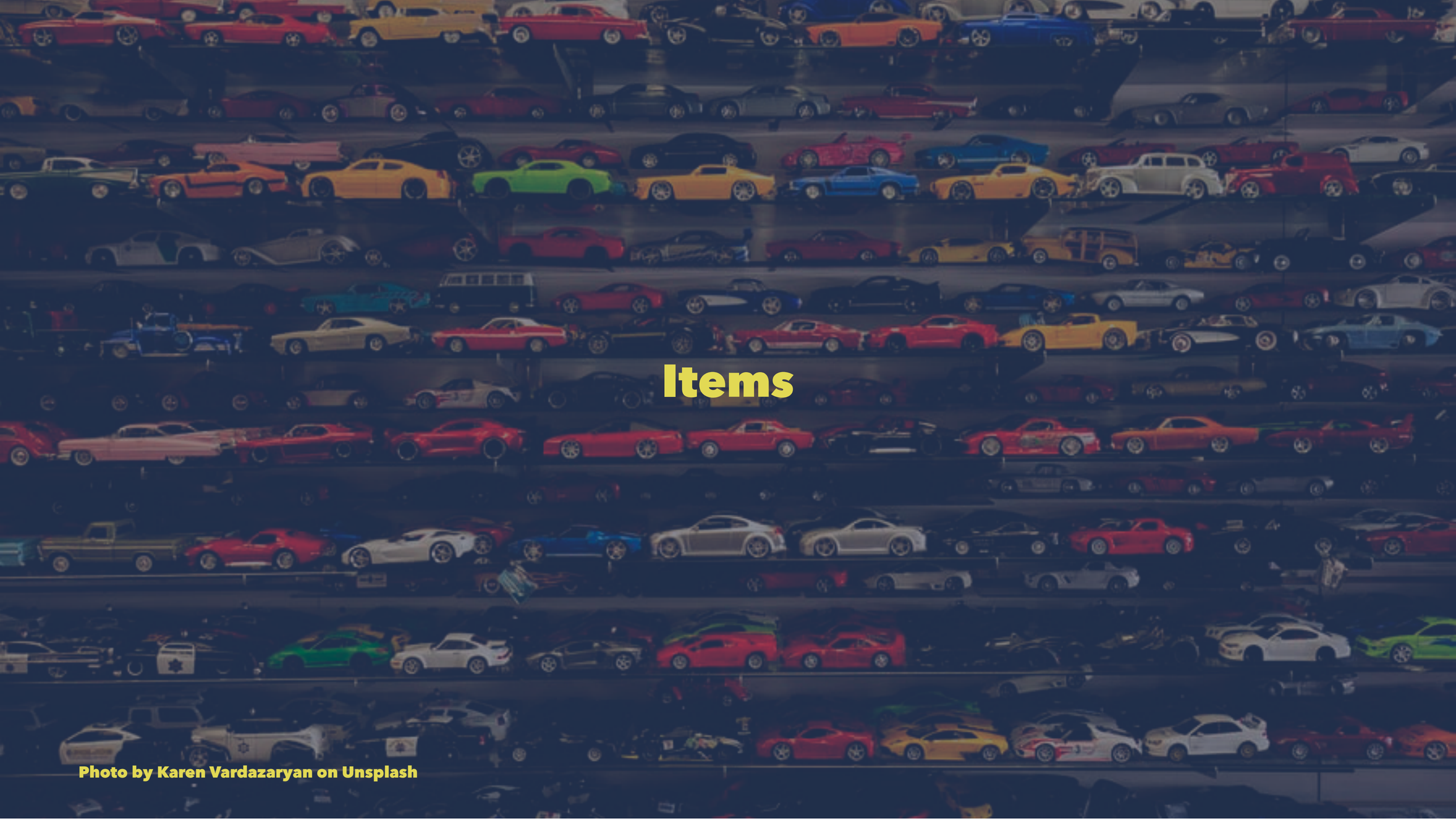


Photo by Karen Vardazaryan on Unsplash

Single data record, or "row" in a table

Each item is uniquely defined by a Primary Key

Attributes (Values)

- Scalars
  - Single items
- Collections
  - Groups of values

Each Item is composed of a group of Attributes

You can specify a single value or several

Scalar Values

- `Binary`
- `Boolean`
- `Number`
- `Null`
- `String`

Collection Values

1. Binary Set
2. List
3. Map
4. Number Set
5. String Set

Primary Key

1. Simple Primary Key – Hash Key
2. Composite Primary Key
    1. Hash Key
    2. Sort Key

Each item in a table is uniquely identified by a primary key.

There are two types.

A simple primary key made up of just a partition key. Similar to a "normal" KV store like Memcache.

A composite primary key made up of a partition key and a sort key. Enables complex queries.

Scalars as Primary Keys

1. String
2. Numbers
3. Binary

These can all be used as primary keys

Data Structure – `ServiceLock`

1. 🌍 `ServiceName ( String )` – Hash Key
2. 📜 `Version ( Int )` – Sort Key
3. 🔴 `IsIncidentOngoing ( Bool )`
4. 🙋 `Username ( String )`
5. ⌚ `Timestamp( String )`
6. ➡️ `Message ( String )`

```
resource "aws_dynamodb_table" "emergency_stop_service_lock" {
  name           = "emergency-stop-service-lock"
  read_capacity  = 10
  write_capacity = 10
  hash_key       = "ServiceName"
  range_key      = "Version"

  attribute {
    name = "ServiceName"
    type = "S"
  }

  attribute {
    name = "Version"
    type = "N"
  }
}
```

We're using Terraform to create a "resource"— a DynamoDB Table.

It's named `emergency-stop-service-lock`

Note the `hash_key` and `range_key` (or `sort_key` make up our composite Primary Key)

We set read/write capacity instead of specifying an instance size or anything like that.

This is very low write volume, and thanks to caching in the application relatively low read volume as well.

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed

So as an example, we can see the first value (at `Version` **1**) shows there was a problem, so the Emergency Stop was pressed.

It must've been a doozy, since it was fixed 2 days later.

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed

When you're querying, you can **only** access this by the composite primary key.

I'm not going to talk about it this time, but if you wanted "all times Joe hit the button" you would create a `Secondary Index` to track that.

Secondary Indexes provide an additional "view" or "rotation" on the data to enable additional access patterns.

Unlike most relational databases, DynamoDB can scale "horizontally" assuming you plan your data model correctly

Note that you need to pick your Primary Key well, that's what Dynamo uses to determine which nodes host your data.

Choosing poorly means you'll have "hot shards" that are overloaded by too many requests!

To learn more, I **highly** recommend *DynamoDB, explained.* by Alex DeBrie.

AWS SDK Swift

- Created by Yuki (**@noppoMan**)
- Major contributions from **@jonnymacs** and **@adam-fowler**

Photo by Levi Morsy on Unsplash

The Core solves authentication, encoding/decoding, an HTTP client, and error propagation.

The second part is the "main" SDK, which contains a Code Generator and the actual APIs & request/response Shapes

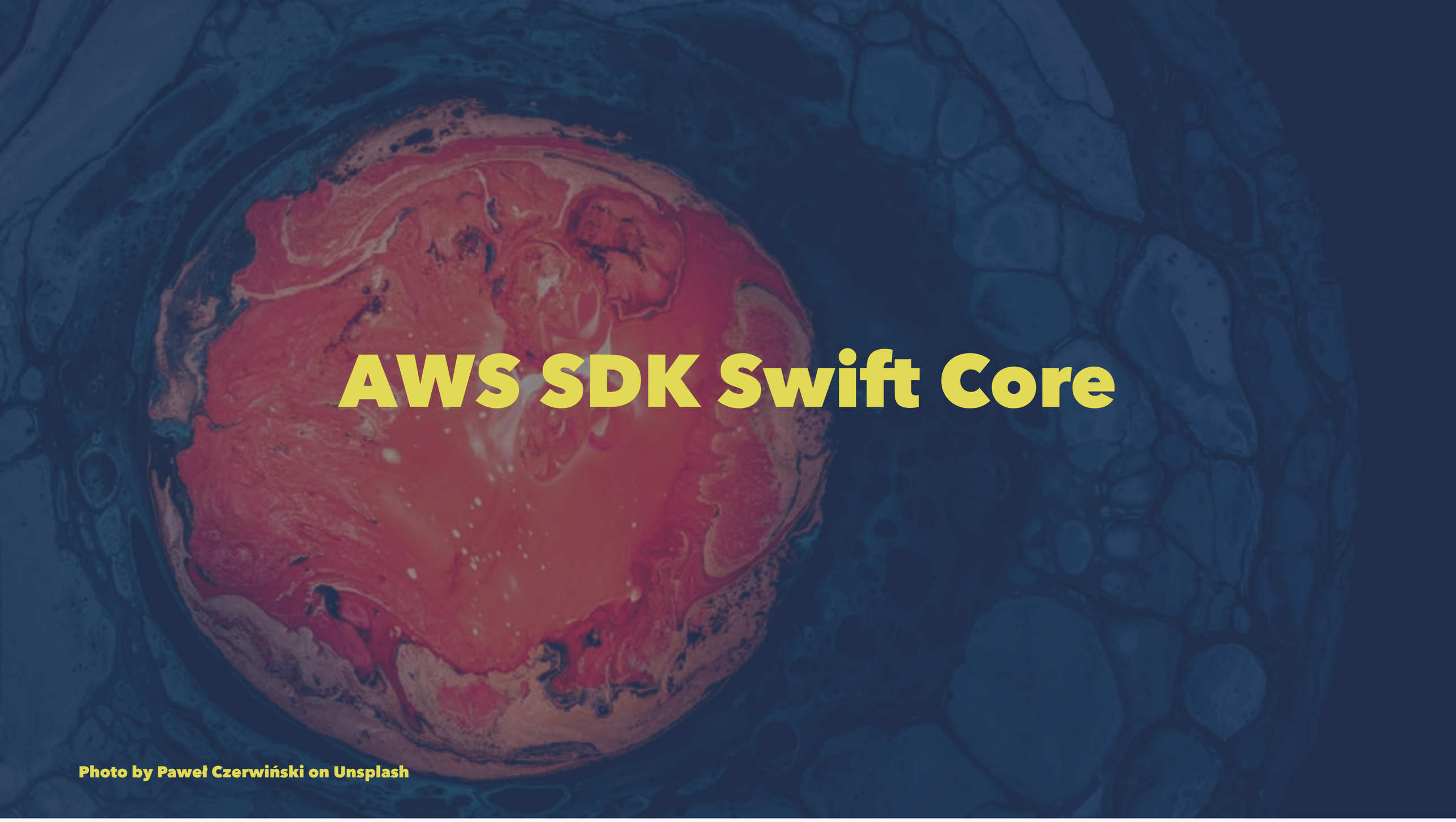


Photo by Paweł Czerwiński on Unsplash

We're going to walk through some of the functionality this provides, learning heavily from other language implementations.

CredentialProvider

```
/// Protocol defining requirements for object providing AWS credentials
public protocol CredentialProvider {
    var accessKeyId: String { get }
    var secretAccessKey: String { get }
    var sessionToken: String? { get }
    var expiration: Date? { get }
}
```

The first thing you'll need to do is have credentials with appropriate permissions to make your API calls!

Three ways to plug in your credentials.

Environment Variables
INI File (`~/.aws`)
Metadata Service (IAM or ECS Profile), which is the best practice.

AWS SDK Swift Core – Hashing

- 💻 `CommonCrypto`
  or
- 🐡 `CAWSSDKOpenSSL`

1. `hmac` – hash-keyed message authentication code
2. `sha256`
3. `md5`

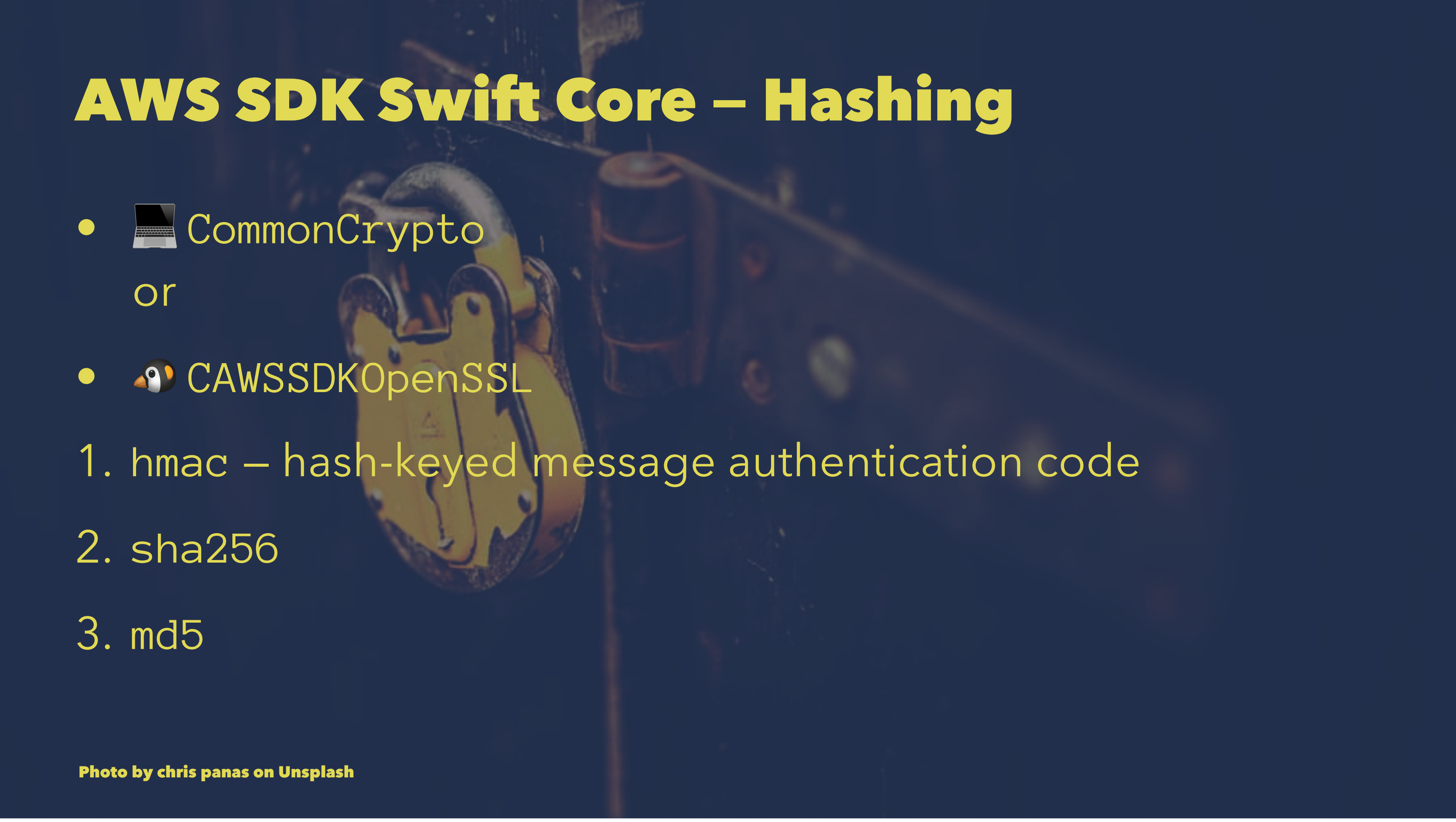


Photo by chris panas on Unsplash

In order to properly sign and authenticate our requests, we need some cyptographic functionality.

We pick `CommonCrypto` if it's available on Darwin platforms, otherwise look for OpenSSL or LibreSSL.

We need an `hmac` function for nearly each part of the request signature, as well as `sha256` for the body and `md5`for S3.

AWS SDK Swift Core – `HTTPCLient`

```
public final class HTTPClient {
    public struct Request {
        var head: HTTPRequestHead
        var body: Data = Data()
    }

    public struct Response {
        let head: HTTPResponseHead
        let body: Data
        public func contentType() -> String?
    }

    public enum HTTPError: Error {
        case malformedHead, malformedBody, malformedURL
    }

    public init(url: URL, eventGroup: EventLoopGroup)

    public func connect(_ request: Request) -> EventLoopFuture<Response>

    public func close(_ callback: @escaping (Error?) -> Void)
}
```

We have our own low-level HTTP client.

We've considered migrating to the `AsyncHTTPClient`, but so far we've appreciated having the fine-grained control of this client we've been using for about a year or so.

```
public class AWSClient {
    public enum RequestError: Error {
        case invalidURL(String)
    }

    public var endpoint: String

    public static let eventGroup: EventLoopGroup

    public func signURL(url: URL,
                        httpMethod: String,
                        expires: Int = 86400) -> URL
...
```

The HTTPClient is wrapped by an `AWSClient`, which is the workhorse of the SDK.

```
public func send<Output: AWSShape, Input: AWSShape>(
    operation operationName: String,
    path: String,
    httpMethod: String,
    input: Input) -> Future<Output> {
        return signer.manageCredential().thenThrowing { _ in
            let awsRequest = try self.createAWSRequest(
                operation: operationName,
                path: path,
                httpMethod: httpMethod,
                input: input
            )
            return try self.createNioRequest(awsRequest)
        }.then { nioRequest in
            return self.invoke(nioRequest)
        }.thenThrowing { response in
            return try self.validate(operation: operationName, response: response)
        }
}
```

When we want to make an API request, the first thing we're going to do is ensure we have credentials. After that succeeds, we will use our input values to construct and sign the request with our credentials.

```
public func send<Output: AWSShape, Input: AWSShape>(
    operation operationName: String,
    path: String,
    httpMethod: String,
    input: Input) -> Future<Output> {
        return signer.manageCredential().thenThrowing { _ in
            let awsRequest = try self.createAWSRequest(
                operation: operationName,
                path: path,
                httpMethod: httpMethod,
                input: input
            )
            return try self.createNioRequest(awsRequest)
        }.then { nioRequest in
            return self.invoke(nioRequest)
        }.thenThrowing { response in
            return try self.validate(operation: operationName, response: response)
        }
}
```

When that processing completes, we convert that into a HTTP request which will get sent over the wire thanks to NIO.

As soon as we get a response back from AWS, we'll decode the message (using the right decoder depending on the service!) and send that back to the user, wrapped in a Future.

```
public func send<Output: AWSShape, Input: AWSShape>(
    operation operationName: String,
    path: String,
    httpMethod: String,
    input: Input) -> Future<Output> {
        return signer.manageCredential().thenThrowing { _ in
            let awsRequest = try self.createAWSRequest(
                operation: operationName,
                path: path,
                httpMethod: httpMethod,
                input: input
            )
            return try self.createNioRequest(awsRequest)
        }.then { nioRequest in
            return self.invoke(nioRequest)
        }.thenThrowing { response in
            return try self.validate(operation: operationName, response: response)
        }
}
```

If it is not successful then `AWSClient` will throw an `AWSErrorType`.

AWS SDK Swift Dynamo Client

1. `AttributeValue`
2. `getItem` / `putItem`
3. `query`

AttributeValue

```
/// An attribute of type Binary. For example:
/// "B": "dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk"
public let b: Data?

/// An attribute of type Boolean. For example:  "BOOL": true
public let bool: Bool?

/// An attribute of type Number. For example:  "N": "123.45"
/// Numbers are sent across the network to DynamoDB as strings,
/// to maximize compatibility across languages and libraries.
/// However, DynamoDB treats them as number type attributes
/// for mathematical operations.
public let n: String?

/// An attribute of type Null. For example:  "NULL": true
public let null: Bool?

/// An attribute of type String. For example:  "S": "Hello"
public let s: String?
```

Calling out that Number is *not* `Numeric`, but instead passed to Dynamo as a `String`!

AttributeValue

```swift
/// An attribute of type Binary Set. For example:
/// "BS": ["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
public let bs: [Data]?

/// An attribute of type List. For example:
/// "L": [ {"S": "Cookies"} , {"S": "Coffee"}, {"N", "3.14159"}]
public let l: [AttributeValue]?

/// An attribute of type Map. For example:
/// "M": {"Name": {"S": "Joe"}, "Age": {"N": "35"}}
public let m: [String: AttributeValue]?

/// An attribute of type Number Set. For example:
/// "NS": ["42.2", "-19", "7.5", "3.14"]
public let ns: [String]?

/// An attribute of type String Set. For example:
/// "SS": ["Giraffe", "Hippo" ,"Zebra"]
public let ss: [String]?
```

Fetching a Single Item

```
///  The GetItem operation returns a set of attributes
/// for the item with the given primary key.
public func getItem(_ input: GetItemInput) -> Future<GetItemOutput> {
    return client.send(operation: "GetItem",
                       path: "/",
                       httpMethod: "POST",
                       input: input)
}
```

This is very much like your traditional key-value store; the `input` contains a `key` to correspond with the primary key you want to retrieve.

If there is no matching item, GetItem does not return any data and there will be no Item element in the response.

GetItemInput

```
/// Determines the read consistency model
public let consistentRead: Bool?

/// A map representing the primary key of the item to retrieve.
public let key: [String: AttributeValue]

/// The name of the table containing the requested item.
public let tableName: String
```

This is a truncated view of this `GetItemInput`

GetItem provides an eventually consistent read by default, so you may get stale data.

If your application requires a strongly consistent read, set ConsistentRead to true.

Although a strongly consistent read might take more time than an eventually consistent read, it always returns the last updated value.

Setting Items

```
///  Creates a new item, or replaces an old item with a new item.
public func putItem(_ input: PutItemInput) -> Future<PutItemOutput> {
    return client.send(operation: "PutItem",
                       path: "/",
                       httpMethod: "POST",
                       input: input)
}
```

If an item that has the same primary key as the new item already exists in the specified table, the new item completely replaces the existing item.

You can perform a conditional put operation (add a new item if one with the specified primary key doesn't exist), or replace an existing item if it has certain attribute values.

Setting Items

```
///  Creates a new item, or replaces an old item with a new item.
public func putItem(_ input: PutItemInput) -> Future<PutItemOutput> {
    return client.send(operation: "PutItem",
                       path: "/",
                       httpMethod: "POST",
                       input: input)
}
```

Conditional Expressions allow you to `put` an item only if its current value satisfies a constraint.

Use them to prevent overwriting with the `attribute_not_exists` conditional expression

PutItemInput

```
/// A map of attribute name/value pairs.
public let item: [String: AttributeValue]

/// Use ReturnValues if you want to get the item attributes
/// as they appeared before they were updated
public let returnValues: ReturnValue?

/// The name of the table to contain the item.
public let tableName: String
```

You must include the Primary key, but you can add many more attributes if necessary

Some rules: No `null` values, `String` & `Binary` must be longer than zero-length, and no empty values

You can return the item's attribute values in the same operation, using the ReturnValues parameter.

Querying for Multiple Items

```
///  The Query operation finds items based on primary key values.
public func query(_ input: QueryInput) -> Future<QueryOutput> {
    return client.send(operation: "Query",
                       path: "/",
                       httpMethod: "POST",
                       input: input)
}
```

You can query any table or secondary index that has a composite primary key (a partition key and a sort key).

A Query operation always returns a result set. If no matching items are found, the result set will be empty.

Query results are always sorted by the sort key value.

You can optionally narrow the scope of the Query operation by specifying a sort key value and a comparison operator in KeyConditionExpression.

```
/// One or more substitution tokens for attribute names
/// in an expression.
public let expressionAttributeNames: [String: String]?

/// One or more values that can be substituted in an expression.
public let expressionAttributeValues: [String: AttributeValue]?

/// The condition that specifies the key values for items to
/// be retrieved by the Query action.
public let keyConditionExpression: String?
```

Use the KeyConditionExpression parameter to provide a specific value for the partition key. The Query operation will return all of the items from the table or index with that partition key value.

```
let nowDate = Date()
let earlierDate = now.addingTimeInterval(-60.0 * 60.0)

let now = formatter.string(from: nowDate)
let earlier = formatter.string(from: earlierDate)

let expressionAttributeNames = [
        "#S": "ServiceName",
        "#T": "Timestamp"]
let expressionAttributeValues = [
        ":global": DynamoDB.AttributeValue(s: "global"),
        ":then": DynamoDB.AttributeValue(s: earlier),
        ":now": DynamoDB.AttributeValue(s: now)]

let keyConditionExpression = "#S = :global AND #T BETWEEN :then AND :now"
```

To walk us through an example, let's say we had a Seondary Index setup on `Timestamp`

We want to query for all the updates made within the last 60 minutes

We start off by defining what *names* we plan to use in our query, in this case `ServiceName` and `Timestamp`

Then we insert the values, we use the ISO 8601 date format.

Finally, we put that together in a `Key Condition` which we use to describe our constraints.

AWS SDK Swift Dynamo Client

1. `AttributeValue`
2. `getItem` / `putItem`
3. `query`

There are some other, simpler APIs for Batch gets/puts, but that's the high-level overview.

However, now that we know what we do about the behavior of `PutItem`, that presents us with a problem.

Photo by Daniel H. Tong on Unsplash

When we're making changes to the button, we want to be able to maintain a historical log of changes over time.

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed

This looks great, but... we don't actually have a way to get "latest value" without doing a full table scan!

And if we remove the `Version` field entirely, updates will overwrite the value

Using that data model is fine for a single value you don't want or need the history for this

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed

This provides a challenge— when we think of something that we want a log, how could we represent this?

In SQL, you could just have the DB generate a new ID for you

Then when you are looking for the current state, look at the latest value

There's a setup for that with Dynamo, and Amazon has a pattern they suggest for this.

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message	Current
global	0	true	Joe	8/27/2019, 7:13:40 PM	Fixed	2
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!	
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed	

Here you notice we've added a new field, as well as a new "default" row at 0.

The row with `Version = 0` is special, it's *always* considered the latest version. It's also the only item with a value for `Current`; that's a pointer to the latest row.

Service Name	Version	Is Incident Ongoing	User name	Time stamp	Message	Current
global	0	true	Joe	8/27/2019, 7:13:40 PM	Fixed	2
global	1	false	Jim	8/25/2019, 2:04:31 AM	Error!	
global	2	true	Joe	8/27/2019, 7:13:40 PM	Fixed	

When we want to make an update, we can perform a conditional `Put` on version `0` to only increment the version if the value is still 2.

We can also use a transaction to add row 3 at the same time.

Denormalized Data

So the lesson I learned here was to focus on how to utilize the structures available to create an appropriate access pattern.

With a SQL database, you're going to create separate patterns that require completely different setups.

With Dynamo, whether you creating an additional Secondary Index or changing your data pattern, you're considering how your application will read your data ahead of time.

Once you set the data access patterns, they're largely locked in stone unless you add another Index (which has limits)

Fluent DynamoDB

Fluent 3 Concepts

1. `Database`
2. `DatabaseConnection`
3. `Provider`

📦 Database

```
public protocol Database {

    associatedtype Connection: DatabaseConnection

    /// Creates a new `DatabaseConnection` that will perform
    /// async work on the supplied `Worker`.
    func newConnection(on worker: Worker) -> Future<Connection>
}
```

Our database manages our connection.

It also creates the AWS SDK client to Dynamo, since it receives the configuration values.

```
public final class DynamoDatabase: Database {
    public typealias Connection = DynamoConnection
    private let config: DynamoConfiguration

    public init(config: DynamoConfiguration) {
        self.config = config
    }

    internal func openConnection() -> DynamoDB {
        return DynamoDB(accessKeyId: config.accessKeyId,
                        secretAccessKey: config.secretAccessKey,
                        region: config.region,
                        endpoint: config.endpoint)
    }

    public func newConnection(on worker: Worker) -> EventLoopFuture<DynamoConnection> {
        do {
            let conn = try DynamoConnection(database: self, on: worker)
            return worker.future(conn)
        } catch {
            return worker.future(error: error)
        }
    }
```

Photo by israel palacio on Unsplash

The DatabaseConnection is what I'd consider the core to actually work with your underlying db. This takes the AWS client library and makes the proper calls based on the query.

```
public protocol DatabaseConnection: DatabaseConnectable, Extendable {
    /// This connection's associated database type.
    associatedtype Database: DatabaseKit.Database
        where Database.Connection == Self

    /// If `true`, this connection has been closed and is no
    /// longer valid.
    /// This is used by `DatabaseConnectionPool` to prune
    /// inactive connections.
    var isClosed: Bool { get }

    /// Closes the `DatabaseConnection`.
    func close()
}
```

```
public final class DynamoConnection: DatabaseConnection {
    public typealias Database = DynamoDatabase

    public var isClosed: Bool {
        return self.handle.isClosed()
    }

    public func close() {
        self.handle.close()
    }

    /// Reference to parent `DynamoDatabase` that created this connection.
    private let database: DynamoDatabase

    internal private(set) var handle: DynamoDB!

    internal init(database: DynamoDatabase, on worker: Worker) throws {
        self.database = database
        self.eventLoop = worker.eventLoop
        self.handle = database.openConnection()
    }
}
```

For us, we can make sure the underlying EventLoop is closed for our DynamoDB client.

Photo by Phix Nguyen on Unsplash

```
public struct DatabaseIdentifier<D: Database>: Equatable,
    Hashable, CustomStringConvertible, ExpressibleByStringLiteral {
    /// The unique id.
    public let uid: String

    /// See `CustomStringConvertible`.
    public var description: String {
        return uid
    }

    /// Create a new `DatabaseIdentifier`.
    public init(_ uid: String) {
        self.uid = uid
    }

    /// See `ExpressibleByStringLiteral`.
    public init(stringLiteral value: String) {
        self.init(value)
    }
}
```

We need a constant which defines the name of each particular database, such as SQLite, MySQL, Postgres, or...

```
extension DatabaseIdentifier {
    /// Default identifier for `DynamoDatabase`.
    public static var dynamo: DatabaseIdentifier<DynamoDatabase> {
        return "dynamo"
    }
}
```

Dynamo

```
/// Capable of creating connections to identified databases.
public protocol DatabaseConnectable: Worker {

    func databaseConnection<Database>(
        to database: DatabaseIdentifier<Database>?) ->
        Future<Database.Connection>

}
```

This is typically going to be the `Request` object that you get once you receive a request.

Photo by Kirill Pershin on Unsplash

```
public protocol DatabaseQueryable {

    associatedtype Query

    associatedtype Output

    /// Asynchronously executes a query passing zero
    /// or more output to the supplied handler.
    func query(_ query: Query,
               _ handler: @escaping (Output) throws -> ()) ->
        Future<Void>
}
```

```swift
public enum DynamoQueryAction {
    case set, get, delete, filter
}

/// 🔎 A DynamoDB operation
public struct DynamoQuery {
    /// 🏹 Note this is a var so `get/set` can be flipped easily if desired
    public var action: DynamoQueryAction

    /// 🍴 The name of the table in DynamoDB to work on
    public let table: String

    /// 💸 Which value(s) to perform the action upon
    public let keys: [DynamoValue]
}
```

For FluentDynamoDB it is the wrapper that allows us to query.

```swift
/// 📖 Submit request to DynamoDB for one value
///
/// The Future signals completion, and the handler will run upon success.
/// Note for .set and .delete queries, the handler will be called with the *old* values
/// that have been replaced!
public func query(_ query: Query, _ handler: @escaping (Output) throws -> ()) -> Future<Void> {
    self.logger?.record(query: String(describing: query))
    do {
        if query.keys.count != 1 {
            throw DynamoConnectionError.improperlyFormattedQuery("`DynamoQuery.keys` should only set one key when requesting a single value.")
        }
        guard let requestedKey = query.keys.first?.encodedKey else {
            throw DynamoConnectionError.improperlyFormattedQuery("`DynamoQuery.keys` should only set one `DynamoValue` when requesting a single value.")
        }
        switch query.action {
        case .get:
            let inputItem = DynamoDB.GetItemInput(
                key: requestedKey, tableName: query.table)
            return self.handle.getItem(inputItem).map { output in
                return try handler(Output(attributes: output.item))
            }
        case .set:
            let inputItem = DynamoDB.PutItemInput(item: requestedKey, returnValues: .allOld, tableName: query.table)
            return self.handle.putItem(inputItem).map { output in
                return try handler(Output(attributes: output.attributes))
            }
        case .delete:
            let inputItem = DynamoDB.DeleteItemInput(
                key: requestedKey, returnValues: .allOld, tableName: query.table)
            return self.handle.deleteItem(inputItem).map { output in
                return try handler(DynamoValue(attributes: output.attributes))
            }
        case .filter:
            return self.eventLoop.newFailedFuture(error: DynamoConnectionError.notImplementedYet)
        }
    } catch {
        return self.eventLoop.newFailedFuture(error: error)
    }
}
```

```
case .set:
    let inputItem = DynamoDB.PutItemInput(item: requestedKey,
                                          returnValues: .allOld,
                                          tableName: query.table)

     return self.handle.putItem(inputItem).map { output in
        return try handler(DynamoValue(attributes: output.attributes))
     }
```

😻 Provider

```
public protocol Provider {
    /// Register all services you would like to provide the `Container` here.
    ///
    ///     services.register(RedisCache.self)
    ///
    func register(_ services: inout Services) throws

    /// Called before the container has fully initialized.
    func willBoot(_ container: Container) throws -> Future<Void>

    /// Called after the container has fully initialized and after `willBoot(_:)`.
    func didBoot(_ container: Container) throws -> Future<Void>
}
```

`Providers` allow third-party services to be easily integrated into a service `Container`.

Providers first register, and have access to a `Services` struct they can mutate. After all providers register, there's a boot phase.

Most of your "real work" that isn't registering related services should be in `didBoot`

```swift
/// 💧 The Provider expected to be registered to easily allow
/// usage of DynamoDB from within a Vapor application
public struct FluentDynamoDBProvider: Provider {
    public func register(_ services: inout Services) throws {
        try services.register(FluentProvider())

        services.register(DynamoConfiguration.self)
        services.register(DynamoDatabase.self)
        var databases = DatabasesConfig()
        databases.add(database: DynamoDatabase.self, as: .dynamo)
        services.register(databases)
    }
    public func didBoot(_ container: Container) throws -> EventLoopFuture<Void> {
        return .done(on: container)
    }
}
```

We want to make sure both the configuration and Database are available during configuration.

We also want to add Dynamo as an available DB.

Note that we're using the `.dynamo` Database Identifier.

```
try services.register(FluentDynamoDBProvider())

var databases = DatabasesConfig()

let credentialsPath = Environment.get("CREDENTIALS_FILENAME") ?? "/etc/emergency-stop.json"
let creds = awsCredentials(path: credentialsPath)
let endpoint = Environment.get("ENVIRONMENT") == "local" ? "http://localhost:8000" : nil

let dynamoConfiguration =  DynamoConfiguration(accessKeyId: creds.accessKey,
                                               secretAccessKey: creds.secretKey,
                                               endpoint: endpoint)


let dynamo = DynamoDatabase(config: dynamoConfiguration)
databases.add(database: dynamo, as: .dynamo)
services.register(databases)
```

```
/// Write a ServiceLock to DynamoDB
///
/// - Returns:
///     An `EventLoopFuture` used to indicate success or failure
public func write(on worker: Request) -> EventLoopFuture<[DynamoValue]> {

    let key = self.dynamoFormat()

    let query = DynamoQuery(action: .set,
                            table: "limit-break-emergency-stop",
                            keys: [key])

    return worker.databaseConnection(to: .dynamo)
        .flatMap { connection in
            connection.query(query)
        }
}
```

Photo by Seth Reese on Unsplash

Next

1. Publish `AWSSDKSwift` 4.0.0
    1. Uses `NIO` 2.0 🎉
2. Convert `FluentDynamoDB` to NIO 2 & Fluent 4
3. Upgrade `DynamoModel`
4. Create `Migration` support (mostly for testing)
5. Better integration with `swift-log` and `swift-metrics`

Thank you!

- 🔴 Emergency Stop
- 📦 DynamoDB
- 🌴 `AWSSDKSwift`
- 💧 Fluent DynamoDB

https://github.com/Yasumoto/fluent-dynamodb