Implementing Beam SDKs

A Deep Dive Into the Swift SDK

Byron Ellis
bce@apache.org

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Isn’t that for writing iOS applications?

BEAM SUMMIT

Because I Wanted To

BEAM SUMMIT

Surprisingly Good “Server Side” Language

- OS support for macOS, Linux and Windows for x86 and ARM
- Flexible syntax well-suited to domain specific languages
  - SwiftUI for implementing GUIs
  - Vapor for implementing Web applications
  - Extensions and tail closure syntax allow for pandas-like API syntax.
- Compiled language with dynamic language features inherited from ObjectiveC
  - C compatible memory layout with pretty good interoperability
  - C++ too, but less developed
- Good library support for Protobuf and gRPC

BEAM SUMMIT

Let’s Dive In

```
try await Pipeline { pipeline in
        let contents = pipeline
            .create(["dataflow-samples/shakespeare"])
            .map { value in
                let parts = value.split(separator: "/", maxSplits: 1)
                print("Got filename \(parts) from \(value)")
                return KV(parts[0].lowercased(), parts[1].lowercased())
            }
            .listFiles(in: GoogleStorage.self)
            .readFiles(in: GoogleStorage.self)

        // Simple ParDo that takes advantage of enumerateLines.
        let lines = contents.pstream { contents, lines in
            for await (content, ts, w) in contents {
                String(data: content, encoding: .utf8)!.enumerateLines { line, _ in
                    lines.emit(line, timestamp: ts, window: w)
                }
            }
        }
        // Our first group by operation
        let baseCount = lines
            .flatMap { (line: String) in line.components(separatedBy: .whitespaces) }
            .groupBy { ($0, 1) }
            .sum()
        let normalizedCounts = baseCount.groupBy {
            ($0.key.lowercased().trimmingCharacters(in: .punctuationCharacters),
             $0.value ?? 1)
        }.sum()
        normalizedCounts.log(prefix: "COUNT OUTPUT")
    }.run(PortableRunner(loopback: true))
```

Protocols let us use a “pandas-like” syntax by attaching new functions to PCollection.

“pstream” is the base user “pardo” construct

BEAM SUMMIT

The Core: PipelineTransform

PipelineTransform represents all conversions from one PCollection to zero or more other PCollections

```
public enum PipelineTransform {
    case pardo(AnyPCollection, String, SerializableFn, [AnyPCollection])
    case impulse(AnyPCollection, AnyPCollection)
    case flatten([AnyPCollection], AnyPCollection)
    case groupByKey(AnyPCollection, AnyPCollection)
    case custom(AnyPCollection, String, Data, Environment?, [AnyPCollection])
    case composite(AnyPCollection, AnyPTransform)
    case external(AnyPCollection, String, FieldValue, [AnyPCollection])
}
```

AnyPCollection is a type-erased PCollection

BEAM SUMMIT

apply() still exists, it's just internal

```
public final class PCollection<Of>: PCollectionProtocol {

@discardableResult
    public func apply(_ transform: PipelineTransform) -> PipelineTransform {
        consumers.append(transform)
        return transform
    }

}

public extension PCollection where Of == Never {

    func impulse() -> PCollection<Data> {
        let output = PCollection<Data>(type: .bounded)
        apply(.impulse(AnyPCollection(self), AnyPCollection(output)))
        return output
    }
}
```

Class in Swift is pass by reference vs Struct which is pass by value

We make heavy use of pattern matching on Of. impulse() is only available on the pipeline root (Never)

This is our impulse implementation

BEAM SUMMIT

For example, implementing FileIO

```
public extension PCollection<KV<String, String>> {

    func readFiles<Source: FileIOSource>(in _: Source.Type) -> PCollection<Data> {
        Source.readFiles(matching: self)
    }

    func listFiles<Source: FileIOSource>(in _: Source.Type) -> PCollection<KV<String, String>> {
        Source.listFiles(matching: self)
    }
}

public protocol FileIOSource {
    static func readFiles(matching: PCollection<KV<String, String>>) -> PCollection<Data>
    static func listFiles(matching: PCollection<KV<String, String>>) -> PCollection<KV<String, String>>
}
```

These will only be available to PCollections of KV<String,String>

Protocols can define static methods

BEAM SUMMIT

Interesting Tidbits…

- We can’t actually serialize closures like you can with Java and Python
  - What we do instead is capture their position within the source code
  - This uniquely identifies them in a given binary without the user having to give them a name
  - This is done via the #fileID and #line macros
- We define pstream on up to three outputs, though we can add more
  - In Swift 6 we should get the ability to iterate over parameter packs and simplify this code
  - This technique is heavily used in Swift (e.g. SwiftUI) so support improves over time
- There is also an element-wise “pardo” that is implemented in terms of “pstream.” It is used to implement “map” and “flatMap”

BEAM SUMMIT

We use Swift async streams to implement ParDo

```
public extension PCollection {

  func pstream<Param: Codable, O0>(name: String, type: StreamType = .unspecified,
                _ param: Param,
                _ fn: @Sendable @escaping (Param, PCollection<Of>.Stream, PCollection<O0>.Stream) async throws -> Void) -> (PCollection<O0>) {
    let output = PCollection<O0>(type: _resolve(self, type))
    apply(.pardo(AnyPCollection(self), name, ParameterizedClosureFn(param, fn), [AnyPCollection(output)]))
    return output
  }

  func reduce<Result: Codable, K, V>(name: String? = nil, _file: String = #fileID, _line: Int = #line,
                into: Result, _ accumulator: @Sendable @escaping (V, inout Result) -> Void) -> PCollection<KV<K, Result>> where Of == KV<K, V>
  {
    pstream(name: name ?? "\(_file):\(_line)", into) { initialValue, input, output in
      for await (kv, ts, w) in input {
        var result = initialValue
        for v in kv.values {
          accumulator(v, &result)
        }
        let intermediate = KV(kv.key, result)
        output.emit(intermediate, timestamp: ts, window: w)
      }
    }
  }

  func sum<K, V: Numeric & Codable>(_ name: String? = nil, _file: String = #fileID, _line: Int = #line) -> PCollection<KV<K, V>> where Of == KV<K, V> {
    reduce(name: name, _file: _file, _line: _line, into: 0) { a, b in b = b + a }
  }

}
```

Inputs and outputs are modeled as asynchronous streams

Single element processing is built on stream processing.

BEAM SUMMIT

Getting back to FileIO…

```
public static func listFiles(matching: PCollection<KV<String, String>>) -> PCollection<KV<String, String>> {
    matching.pstream(type: .bounded) { matching, output in
        guard let tokenProvider = DefaultTokenProvider(scopes: ["storage.objects.list"]) else {
            throw ApacheBeamError.runtimeError("Unable to get OAuth2 token.")
        }
        let connection = Connection(provider: tokenProvider)
        for await (match, ts, w) in matching {
            let bucket = match.key
            for prefix in match.values {
                let response: Data? = try await withCheckedThrowingContinuation { continuation in
                    do {
                        try connection.performRequest(
                            method: "GET",
                            urlString: "https://storage.googleapis.com/storage/v1/b/\(bucket)/o",
                            parameters: ["prefix": prefix],
                            body: nil
                        ) { data, _, error in
                            if let e = error {
                                continuation.resume(throwing: e)
                            } else {
                                continuation.resume(returning: data)
                            }
                        }
                    } catch {
                        continuation.resume(throwing: error)
                    }
                }
                if let data = response {
                    let listfiles = try JSONDecoder().decode(ListFilesResponse.self, from: data)
                    for item in listfiles.items {
                        if item.size != "0" {
                            output.emit(KV(item.bucket, item.name), timestamp: ts, window: w)
                        }
                    }
                }
            }
        }
    }
}
```

BEAM SUMMIT

Building that into a Pipeline…

```
var context: PipelineContext {
  get throws {
    // Grab the pipeline content using an new root
    var root = PCollection<Never>(coder: .unknown(.coderUrn("never")), type: .bounded)
    _ = content(&root)
//…
    while toVisit.count > 0 {

      let item = toVisit.removeFirst()
      if case let .transform(parents, pipelineTransform) = item {
        let inputs = parents.enumerated().map { ("\($0)", "\($1.name)") }.dict()
        switch pipelineTransform {
        case let .pardo(_, n, fn, o):
          let outputs = try o.enumerated().map {
            try ("\($0)", collection(from: $1).name)
          }.dict()
//…
        case let .impulse(_, o):
          let outputs = try [o].enumerated().map {
            try ("\($0)", collection(from: $1).name)
          }.dict()
          let p = try transform { _, name in
            .with {
              $0.uniqueName = name
              $0.outputs = outputs
              $0.spec = .with {
                $0.urn = .transformUrn("impulse")
              }
            }
          }
          rootIds.append(p.name)
          toVisit.append(.collection(o))
```

The “pipeline” is a Never PCollection

Lazily evaluated closure

Breadth-first walk over all transforms and pcollections

BEAM SUMMIT

Eventually resulting in a PipelineContext

```
public final class PipelineContext {
    var proto: PipelineProto
    let defaultEnvironmentId: String
    let collections: [String: AnyPCollection]
    let pardoFns: [String: SerializableFn]
}
```

- Proto is used to submit pipeline to a Runner such as Prism
- ParDoFns and Collections are used by the Worker to constructor BundleProcessors

BEAM SUMMIT

Worker Implementation: Control Plane

```
    public func start() throws {
        let group = PlatformSupport.makeEventLoopGroup(loopCount: 1)
        let client = try Org_Apache_Beam_Model_FnExecution_V1_BeamFnControlAsyncClient(channel: GRPCChannelPool.with(endpoint: control,
eventLoopGroup: group))
        let (responses, responder) = AsyncStream.makeStream(of: Org_Apache_Beam_Model_FnExecution_V1_InstructionResponse.self)
        let options = CallOptions(customMetadata: ["worker_id": id])
        let control = client.makeControlCall(callOptions: options)

        Task {
            for await r in responses {
                try await control.requestStream.send(r)
            }
        }
```

This is the important bit: An asynchronous stream that sends control plane responses from anywhere.

BEAM SUMMIT

This Is All You Really Need....

```
for try await instruction in control.responseStream {

    switch instruction.request {
    case let .processBundle(pbr):
        do {
            let p = try await processor(for: pbr.processBundleDescriptorID)
            let accumulator = MetricAccumulator(instruction: instruction.instructionID, registry: registry)
            await accumulator.start()
            metrics[instruction.instructionID] = accumulator
            Task {
                await p.process(instruction: instruction.instructionID,accumulator: accumulator, responder: responder)
            }
        } catch {
            log.error("Unable to process bundle \(pbr.processBundleDescriptorID): \(error)")
        }
```

We fork off a new Task to handle a bundle

BEAM SUMMIT

Building a Bundle Processor

```
init(id: String,
     descriptor: Org_Apache_Beam_Model_FnExecution_V1_ProcessBundleDescriptor,
     collections: [String: AnyPCollection],
     fns: [String: SerializableFn]) throws
{
    log = Logging.Logger(label: "BundleProcessor(\(id) \(descriptor.id))")
    var temp: [Step] = []
    let coders = BundleCoderContainer(bundle: descriptor)
    var streams: [String: AnyPCollectionStream] = [:]
    // First make streams for everything in this bundle (maybe I could use the pcollection array for this?)
    for (_, transform) in descriptor.transforms {
        for id in transform.inputs.values {
            if streams[id] == nil {
                streams[id] = collections[id]!.anyStream
            }
        }
        for id in transform.outputs.values {
            if streams[id] == nil {
                streams[id] = collections[id]!.anyStream
            }
        }
    }
```

First we construct all the asynchronous streams we need. These are passed to “pstream” calls

BEAM SUMMIT

Building a Bundle Processor

```
    for (transformId, transform) in descriptor.transforms {
        let urn = transform.spec.urn
        // Map the input and output streams in the correct order
        let inputs = transform.inputs.sorted().map { streams[$0.1]! }
        let outputs = transform.outputs.sorted().map { streams[$0.1]! }
…
        if urn == "beam:transform:pardo:v1" {
            let pardoPayload = try Org_Apache_Beam_Model_Pipeline_V1_ParDoPayload(serializedData: transform.spec.payload)
            if let fn = fns[transform.uniqueName] {
                temp.append(Step(transformId: transform.uniqueName,
                                 fn: fn,
                                 inputs: inputs,
                                 outputs: outputs,
                                 payload: pardoPayload.doFn.payload))
            } else {
                log.warning("Unable to map \(transform.uniqueName) to a known SerializableFn. Will be skipped during processing.")
            }
```

We map our functions and parameters to our streams

Everything gets appended to a list of steps

BEAM SUMMIT

Source and Sink Steps are special

```
if urn == "beam:runner:source:v1" {
    let remotePort = try RemoteGrpcPort(serializedData: transform.spec.payload)
    let coder = try Coder.of(name: remotePort.coderID, in: coders)
    log.info("Source '\(transformId)','\(transform.uniqueName)' \(remotePort) \(coder)")
    try temp.append(Step(
        transformId: transform.uniqueName == "" ? transformId : transform.uniqueName,
        fn: Source(client: .client(for: ApiServiceDescriptor(proto: remotePort.apiServiceDescriptor), worker: id), coder: coder),
        inputs: inputs,
        outputs: outputs,
        payload: Data()
    ))
} else if urn == "beam:runner:sink:v1" {
    let remotePort = try RemoteGrpcPort(serializedData: transform.spec.payload)
    let coder = try Coder.of(name: remotePort.coderID, in: coders)
    log.info("Sink '\(transformId)','\(transform.uniqueName)' \(remotePort) \(coder)")
    try temp.append(Step(
        transformId: transform.uniqueName == "" ? transformId : transform.uniqueName,
        fn: Sink(client: .client(for: ApiServiceDescriptor(proto: remotePort.apiServiceDescriptor), worker: id), coder: coder),
        inputs: inputs,
        outputs: outputs,
        payload: Data()
    ))
}
```

Pulls things off of the Data Plane

Puts things on the Data Plane

BEAM SUMMIT

We then spawn discrete tasks for each step

```
_ = await withThrowingTaskGroup(of: (String, String).self) { group in
    log.info("Starting bundle processing for \(instruction)")
    var count: Int = 0
    do {
        // Start metrics handling for this instruction
        for step in steps {
            log.info("Starting Task \(step.transformId)")
            let context = SerializableFnBundleContext(instruction: instruction, transform: step.transformId, payload: step.payload, metrics: MetricReporter(accumulator: accumulator, transform: step.transformId), log: log)
            group.addTask {
                try await step.fn.process(context: context, inputs: step.inputs, outputs: step.outputs)
            }
            count += 1
        }
        var finished = 0
        for try await (instruction, transform) in group {
            finished += 1
            log.info("Task Completed (\(instruction),\(transform)) \(finished) of \(count)")
        }
        await accumulator.reporter.yield(.finish({ metricInfo,metricData in
            log.info("All tasks completed for \(instruction)")
            responder.yield(.with {
                $0.instructionID = instruction
                $0.processBundle = .with {
                    $0.monitoringData.merge(metricData, uniquingKeysWith: {a,b in b})
                    $0.monitoringInfos.append(contentsOf: metricInfo)
                    $0.requiresFinalization = true
                }
            })
        }));
```

Wait for them all to finish here

Report completion of the bundle

BEAM SUMMIT

Source Steps

```
func process(context: SerializableFnBundleContext,
             inputs _: [AnyPCollectionStream], outputs: [AnyPCollectionStream]) async throws -> (String, String)
{
    log.info("Waiting for input on \(context.instruction)-\(context.transform)")
    let bytesRead = await context.metrics.counter(name: "bytes-read")
    let recordsRead = await context.metrics.counter(name: "records-read")
    let (stream, _) = await client.makeStream(instruction: context.instruction, transform: context.transform)
    var messages = 0
    var count = 0
    for await message in stream {
        messages += 1
        switch message {
        case let .data(data):
            var d = data
            var totalBytes = data.count
            var localCount = 0
            while d.count > 0 {
                let value = try coder.decode(&d)
                for output in outputs {
                    try output.emit(value: value)
                    count += 1
                    localCount += 1
                }
            }
            bytesRead(totalBytes)
            recordsRead(localCount)
```

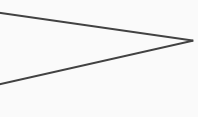
This is a Data Plane stream.

Decode and forward to all outputs

BEAM SUMMIT

Source Steps

```
        case let .last(id, transform):
            for output in outputs {
                output.finish()
            }
            await client.finalizeStream(instruction: id, transform: transform)
            log.info("Source \(context.instruction),\(context.transform) handled \(count) items over \(messages) messages")
            return (id, transform)
        // TODO: Handle timer messages
        default:
            log.info("Unhandled message \(message)")
        }
    }
    return (context.instruction, context.transform)
  }
}
```



Data Plane sends a “last” message when the stream is done.

BEAM SUMMIT

Sink Steps

```
func process(context: SerializableFnBundleContext,
             inputs: [AnyPCollectionStream], outputs _: [AnyPCollectionStream]) async throws -> (String, String)
{
    let bytesWritten = await context.metrics.counter(name: "bytes-written")
    let recordsWritten = await context.metrics.counter(name: "records-written")
    let (_, emitter) = await client.makeStream(instruction: context.instruction, transform: context.transform)
    var bytes = 0
    var records = 0
    for try await element in inputs[0] {
        var output = Data()
        try coder.encode(element, data: &output)
        bytes += output.count
        records += 1
        emitter.yield(.data(output))
    }
    bytesWritten(bytes)
    recordsWritten(records)
    emitter.yield(.last(context.instruction, context.transform))
    emitter.finish()
    await client.finalizeStream(instruction: context.instruction, transform: context.transform)
    return (context.instruction, context.transform)
}
```

We do the reverse: Encode output and send it to the Data Plane

BEAM SUMMIT

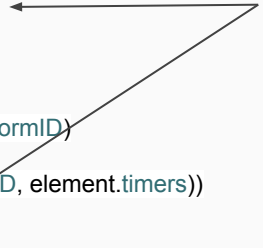
Data Plane

- Multiplexes data from potentially many different bundles into the same channel
- Sends/receives two different types of messages in (potentially) groups
  - Data with a target instruction and transform, may also have a “last message” marker
  - Timers also with a target instruction and transform
- We need to ensure these are distributed to the proper sources
- Coming from Sinks we need to multiplex data back onto the Data Plane

BEAM SUMMIT

Demultiplexing

```
for try await elements in stream.responseStream {
        var last: [Pair: Message] = [:] // Split out last calls so they are always at the end
        var messages: [Pair: [Message]] = [:]
        for element in elements.data {
            let key = Pair(id: element.instructionID, transform: element.transformID)
            if element.data.count > 0 {
                messages[key, default: []].append(.data(element.data))
            }
            if element.isLast {
                last[key] = .last(element.instructionID, element.transformID)
            }
        }
        for element in elements.timers {
            let key = Pair(id: element.instructionID, transform: element.transformID)
            if element.timers.count > 0 {
                messages[key, default: []].append(.timer(element.timerFamilyID, element.timers))
            }
            if element.isLast {
                last[key] = .last(element.instructionID, element.transformID)
            }
        }
```



Want to make sure all “last” events are indeed last

BEAM SUMMIT

Demultiplexing

```
        // Send the messages to registered sources
        for (key, value) in messages {
            let output = await self.makeStream(key: key).1
            for v in value {
                output.yield(v)
            }
        }
        // Send any last messages
        for (key, value) in last {
            let output = await self.makeStream(key: key).1
            output.yield(value)
        }
    }
```

Value is iterable so less expensive than it looks!

Multiplexing

```
Task {
  log.info("Initiating data plane multiplexing.")
  let input = multiplex.0
  var count = 0
  var flushes = 0
  var elements = Org_Apache_Beam_Model_FnExecution_V1_Elements()
  for try await element in input {
    var shouldFlush = false
    switch element.message {
    case let .data(payload):
      elements.data.append(.with {
        $0.instructionID = element.id
        $0.transformID = element.transform
        $0.data = payload
      })
      count += 1
    case let .timer(family, payload):
      elements.timers.append(.with {
        $0.instructionID = element.id
        $0.transformID = element.transform
        $0.timerFamilyID = family
        $0.timers = payload
      })
      count += 1
```

Data and timer messages come from Sink Steps

BEAM SUMMIT

Multiplexing

```
        case let .last(id, transform):
            elements.data.append(.with {
                $0.instructionID = id
                $0.transformID = transform
                $0.isLast = true
            })
            shouldFlush = true
            count += 1
        case .flush:
            shouldFlush = true
        }
        if shouldFlush || elements.data.count + elements.timers.count >= flush {
            do {
                if case .last = element.message {
                    log.info("Got last message, flushing \(elements.data.count + elements.timers.count) elements to data plane")
                }
                try await stream.requestStream.send(elements)
            } catch {
                log.error("Unable to multiplex elements onto data plane: \(error)")
            }
            elements = Org_Apache_Beam_Model_FnExecution_V1_Elements()
            shouldFlush = false
            flushes += 1
        }
        if count % 50000 == 0, count > 0 {
            log.info("Processed \(count) elements (\(flushes) flushes)")
        }
    }
    if elements.data.count + elements.timers.count > 0 {
        do {
            log.info("Flushing final elements to data plane.")
            try await stream.requestStream.send(elements)
        } catch {
            log.error("Unable to multiplex final elements onto data plane: \(error)")
        }
    }
    log.info("Shutting down dataplane multiplexing")
}
```

We also have discrete “last” and “flush” messages that can be propagated by Steps

BEAM SUMMIT

Coding

```
public indirect enum Coder {
    /// Catch-all for coders we don't understand. Mostly used for error reporting
    case unknown(String)

    case custom(Data)
    /// Standard scalar coders. Does not necessarily correspond 1:1 with BeamValue. For example, varint and fixedint both map to integer
    case double, varint, fixedint, byte, bytes, string, boolean, globalwindow
    /// Composite coders.
    case keyvalue(Coder, Coder)
    case iterable(Coder)
    case lengthprefix(Coder)
    case windowedvalue(Coder, Coder)
    /// Schema-valued things
    case row(Schema)
}
```

BEAM SUMMIT

Coding

```
/// An enum representing values coming over the FnApi Data Plane.
public indirect enum BeamValue {
  /// A value not representable in the Swift SDK
  case invalid(String)
  // Scalar values
  /// Bytes coded
  case bytes(Data?)
  /// UTF8 Strings
  case string(String?)
  /// Integers (Signed 64-bit)
  case integer(Int?)
  /// Doubles
  case double(Double?)
  /// Booleans
  case boolean(Bool?)
  /// A window
  case window(Window)
  /// Schema-valued thing. Doesn't technically need to be a row, but that's the only coder support.
  case row(FieldValue)
  // Composite Values
  /// An iterable
  case array([BeamValue])
  /// A key-value pair
  case kv(BeamValue, BeamValue)
  /// A windowed value
  case windowed(BeamValue, Date, UInt8, BeamValue)
```

BEAM SUMMIT

Coding

```
public protocol Beamable {
   static var coder: Coder { get }
}
extension Data: Beamable {
   public static let coder: Coder = .bytes
}
extension String: Beamable {
   public static let coder: Coder = .string
}
extension Int: Beamable {
   public static let coder: Coder = .varint
}
extension Bool: Beamable {
   public static let coder: Coder = .boolean
}
```

BEAM SUMMIT

Schemas

```
public indirect enum FieldValue {
  // Variable width numbers
  case int(Int, FieldType)
  case float(Double, FieldType)
  case decimal(Decimal, FieldType)
  // Other scalar types
  case boolean(Bool)
  case string(String)
  case datetime(Date)
  case bytes(Data)
  case null
  case undefined
  case logical(String, FieldValue)
  case row(Schema, [FieldValue])
  case array([FieldValue])
  case repeated([FieldValue])
  case map([(FieldValue, FieldValue)])
```

Not going to spend too much time here. Very similar to coders, but not similar enough to reuse the code.

BEAM SUMMIT

What Went Well?

- Getting the basic Worker up and running is pretty easy if you have good gRPC support
  - The only “gotcha” is the use of custom metadata to identify the worker.
  - You can get pretty far just implementing process bundle and the dataplane.
- New (at the time) asynchronous stream support made implementing bundle processing very clean
- Developing against Prism is pretty fast and easy
  - Though it would be nice to have some debugging/SDK implementation features like extended logging.

BEAM SUMMIT

What Went Poorly?

- Repository structure made it hard to use “the model.”
  - Current package management though (e.g. Rust, Go, Swift) uses “Git repo” as its unit of dependency
  - Limitations of git make it hard to depend on the Beam repo (e.g. depend on the protobuf)
- Portable local runners are inconsistent
  - Python runner is the most permissive. Trying the Flink runner meant a lot of going back and reimplementing things
  - Prism helps with this now, but a year ago was just becoming available.
- Coders
  - Schemas are close but not quite the same as base Coders so you end up reimplementing nearly identical code

BEAM SUMMIT

Thank you!

Questions?

LinkedIn
https://www.linkedin.com/in/byellis/

BlueSky

@fdaapproved.bsky.social

BEAM SUMMIT

Title and body

You do not really need to use this template for all your slides.

As long as you use the title/cover slide, that is enough. You can use your own template for the rest of your presentation.

But if you want to use this, that is great!

BEAM SUMMIT

This is a section header

BEAM SUMMIT

Title on color

Maybe use this layout to show a square or vertical image on the right?

BEAM SUMMIT