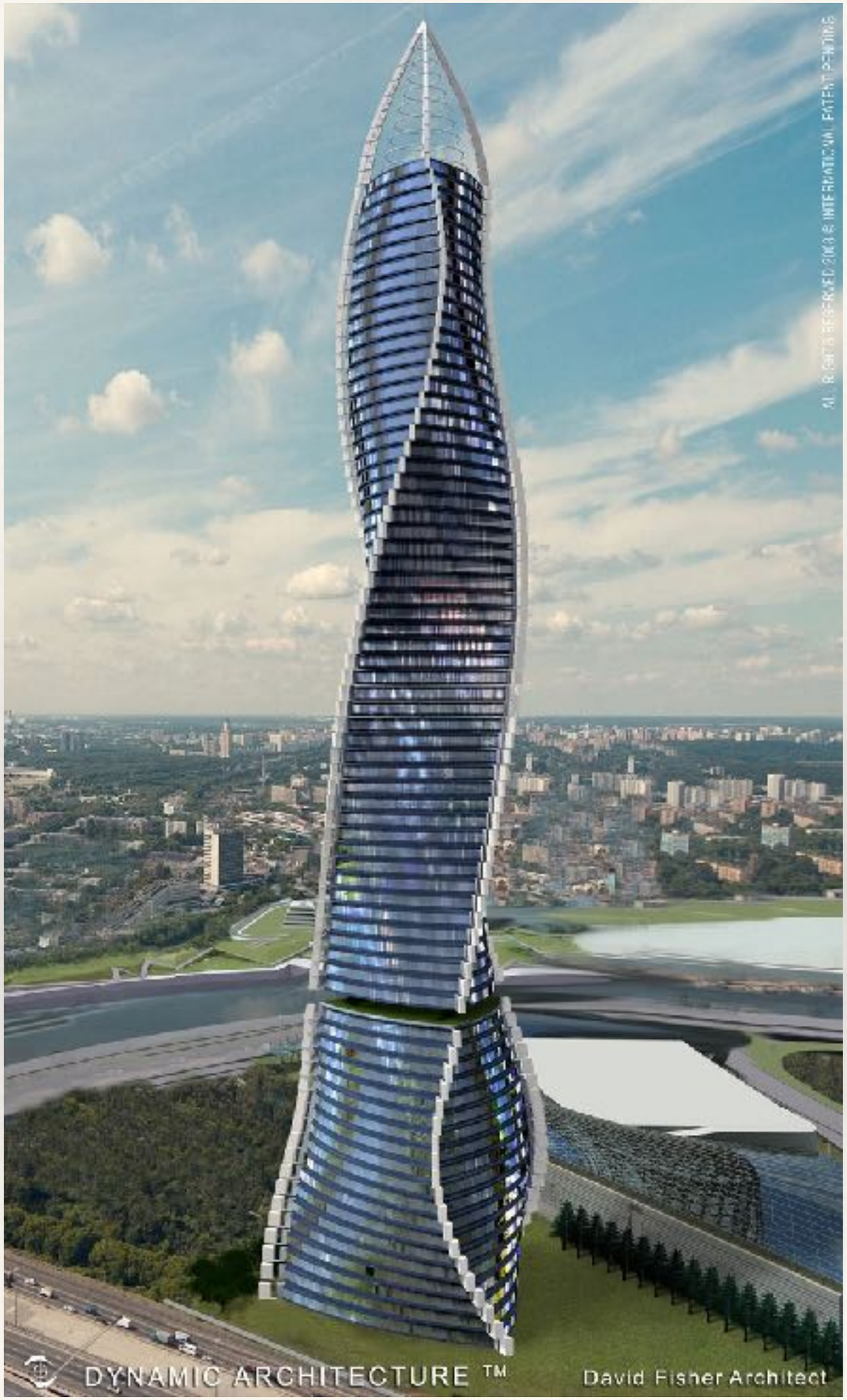
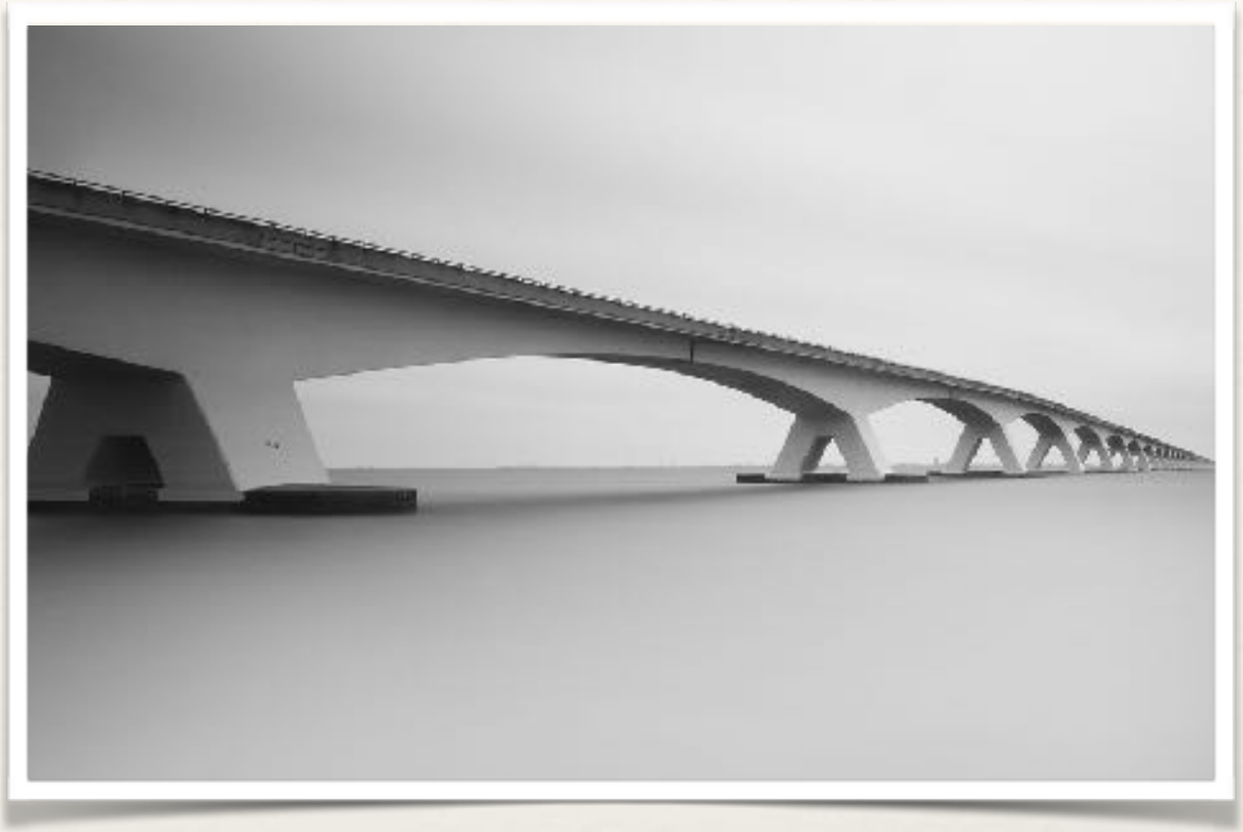
Building a Data Structure in Swift

Patrick Goley
@bitsbetweenbits



Why Though?



- Solve a real world problem
- Explore advance language features
- Look at a robust Swift library project

Configurate

- Library for building configuration objects
  Key-value collections for things like API keys, feature flags, etc

- Allow multiple sources (anything that can act like a key-value collection)

- Configs can be stacked to override other configs

Effective Result

Overrides

Defaults

Architectures

Setting	Resolved	Configurate	Configurate	macOS Default
Base SDK	Latest macOS (macOS 10....		**Latest macOS (macOS 10...**	
Build Active Architecture Only	<Multiple values>		**<Multiple values>**	No
Debug	Yes		**Yes**	No
Release	No			No
Supported Platforms	watchsimulator watchos ma...		**watchsimulator watchos...**	macOS

Build Options

Setting	Resolved	Configurate	Configurate	macOS Default
Debug Information Format	<Multiple values>		**<Multiple values>**	
Debug	DWARF		**DWARF**	
Release	DWARF with dSYM File		**DWARF with dSYM File**	
Enable Testability	**Yes**	**Yes**		No

Deployment

Setting	Resolved	Configurate	Configurate	macOS Default
COMBINE_HIDPI_IMAGES	Yes		**Yes**	
Installation Directory	/Library/Frameworks			
Skip Install	**Yes**	**Yes**		No
Strip Debug Symbols During Copy	<Multiple values>		**<Multiple values>**	Yes
Debug	No		**No**	Yes
Release	Yes		**Yes**	Yes
Strip Linked Product	Yes			Yes
macOS Deployment Target	macOS 10.10		**macOS 10.10**	macOS 10.13

Config Requirements

- Each layer can be queried by key
- The layers can be stacked
- The entire stack can be queried by key
- Layers higher in the stack (added later) are queried first, falling back to lower levels if a value isn't found
- Express requirements as protocols!

Protocols

- Define a set of requirements to be implemented
- associatedtypes require a type to be specified that is used in the protocol

```
protocol MyProtocol {

    associatedtype RelatedType: Equatable

    func niceFunction(arg: RelatedType)
}
```

Conformance

```
struct MyStruct: MyProtocol {

    typealias RelatedType = String

    func niceFunction(arg: String) {
        print(arg)
    }
}

struct MyGenericStruct<T: Equatable>: MyProtocol {

    typealias RelatedType = T

    func niceFunction(arg: T) {
        print(arg)
    }
}
```

Extension

- Define concrete methods on the protocol itself
- Conforming types have these added to their API
- Allows default implementations and shared logic

```
extension MyProtocol {

    func niceArrayFunction(args: [RelatedType]) {

        for arg in args {

            niceFunction(arg: arg)
        }
    }
}
```

KeyedAccessCollection

- Query value for a given key
- Basis for our config objects - allows any source of values

```
public protocol KeyedAccessCollection {

    associatedtype Key: Hashable
    associatedtype Value

    func get(_ key: Key) -> Value?
}
```

Conforming Types

```
extension UserDefaults: KeyedAccessCollection {

    public typealias Key = String
    public typealias Value = Any

    public func get(_ key: String) -> Any? {

        return object(forKey: key)
    }
}

extension ProcessInfo: KeyedAccessCollection {

    public typealias Key = String
    public typealias Value = String

    public func get(_ key: String) -> String? {

        return environment[key]
    }
}
```

Stack

- A Sequence of Elements (allows for loops, map, etc)
- Push to add an Element, pop to remove
- Hold the layers of our configuration object, last pushed configs are inspected first

```
protocol Stack: Sequence {

    mutating func push(_ element: Element)

    mutating func pop() -> Element
}
```

Protocol Composition

- Combine multiple protocols to create a new one
- Enforce constraints on associatedtypes

```
protocol Playlist: Equatable, Sequence where Element == Song {

}

typealias Serializable = Codable & Decodable
```

Putting it all Together

```
protocol KeyedAccessCollectionStack: KeyedAccessCollection, Stack
    where Element: KeyedAccessCollection,
    Element.Key == Key,
    Element.Value == Value {

}
```

Protocol Extension

```
extension KeyedAccessCollectionStack {

    public func get(_ key: Key) -> Value? {

        for keyValueCollection in self {

            if let val = keyValueCollection.get(key) {

                return val
            }
        }

        return nil
    }
}
```

Conforming types now only need to implement Stack

MapStack

- Uses an Array to implement Stack
- Element = AnyKeyedAccessCollection
- Type erasure needed since we can’t have references to generic protocols (existentials)
  More on type erasure

Testing

- Important for data structures to express all behavior and cover edge cases
- Enable coverage reports in your scheme
- View coverage in pull requests!

Name	Coverage
Configurate.framework	99.32%
FoundationConfigTypes.swift	100%
MapStack.swift	95.83%
Dictionary+KeyedAccessCollection.swift	100%
KeyedAccessCollection.swift	100%
ConfigFile.swift	100%
KeyedAccessCollectionStack.swift	100%
Config.swift	100%

Testing... on Linux!

- Make your Swift library useful to server or IOT applications
- Ensure proper functioning against Swift Foundation.
- Use Docker!

Dockerfile

- First install Docker
- Add Dockerfile below
- Execute docker build in the project directory

```
FROM swiftdocker/swift

WORKDIR /package

COPY . ./

RUN swift package resolve
RUN swift package clean
RUN swift test --parallel
```

Other Best Practices

- Execute tests and gather code coverage automatically on Travis CI (free for open source repositories)
- Support package managers: Cocoapods, Swift PM (sorry Carthage)
- Have a license file
- Have open issues!

Links

- https://github.com/patgoley/Configurate
- https://www.docker.com/
- https://travis-ci.org/
- https://codecov.io/

Coding Challenge

- Challenge repository on NashvilleCocoaheads Github
- Implement Sequence with a linked list.
  Make it compile and pass unit tests, you win!
- Submit solutions via pull request
- Happy to help!