From one buzzword to another

Sebastian Osiński

About me

iOS Developer at Tooploox

So... what's with the title?

CQRS

Command Query Responsibility Segregation

Command

Changes state, does not return anything

Query

Returns some data, does not change state

Buzzword #1: Enums

Query

```
enum Query {

    case project(id: String)
    case projects(filters: [String: String])
    case users
    case user(id: String)
    case teams
    case team(id: String)
    .
    .
    .
    var path: String {
        switch self {
        case .project, .projects:
            return "projects"
        case .users:
            return "users"
        case .teams:
            return "teams"
        .
        .
        .
    }

    var parameters: [String: String]? {
        switch self {
            ...
        }
    }

    var urlRequest: URLRequest {
        ... // uses parameters and path to build urlRequest
    }
}
```

Command

```
enum Command {
    case saveDraft(id: String, title: String, description: String)
    case updateDraft(id: String, title: String, description: String)
    case submitProject(id: String)
    case approveProject(id: String)
    case rejectProject(id: String)
    .
    .
    .
    var path: String {
        switch self {
        case saveDraft:
            return "save-draft"
        case updateDraft:
            return "update-draft"
        case submitProject:
            return "submit-project"
        case approveProject:
            return "approve-project"
        case rejectProject:
            return "reject-project"
        .
        .
        .
        }
    }

    var method: String {
        switch self {
            ...
        }
    }

    var body: [String: Any] {
        switch self {
            ...
        }
    }

    var urlRequest: URLRequest {
        ... // uses path, method and body to build urlRequest
    }
}
```

Pros

- All commands / queries are namespaced and easy to find when they need to be used
- Everything in one file

Cons

- Everything in one file 😭

Cons

- To get all info about request, we need to scroll through whole file and visit each `switch`

```
var path: String {
    switch self {
    case approveProject:
        return "approveProject"
    case createProject:
        return "createProject":
    }
}

var method: String {
    switch self {
    case approveProject, createProject:
        return "POST"
    }
}

var body: String {
    switch self {
    case approveProject(let id):
        return ["id": id]
    case createProject(let id, let content):
        return ["id": id, "content": content]
    }
}
```

Cons

- Doesn't scale well - enum grows with each endpoint added

Cons

- Handling similar requests causes switches to grow horizontally

```
var body: String {
    switch self {
    case approveProject(let id), rejectProject(let id), removeProject(let id):
        return ["id": id]
    ...
    }
}
```

Cons

- You can't declare return type of Query

Buzzword #2: POP

Key components:

- Base protocols with default implementations for creating URLRequest
- Specialized protocols for common types of Commands / Queries
- Each command / query is defined as a simple struct

Base protocols: Command

```
protocol Command {
    static var path: String { get }
    static var method: CommandMethod { get } // .post, .put, .delete, .patch

    var bodyDict: JSON { get } // typealias JSON = [String: Any]
}

extension Command {
    var urlRequest: URLRequest {
        // generates proper urlRequest using `path`, `method` and `bodyDict`
        ...
    }
}
```

Base protocols: Query

```
protocol Query {
    associatedtype Result: Decodable
    static var path: String { get }

    var parameters: [String: String]? { get }
}

extension Query {
    var urlRequest: URLRequest {
        // always GET
        // generates proper urlRequest using `path` and `parameters`
        ...
    }
}
```

```
class ApiClient {

    private let session = URLSession.shared
    private let jsonDecoder = JSONDecoder()

    func execute(_ command: Command,
                   success: CommandSuccessCallback?,
                   failure: FailureCallback?) {
        let task = session.dataTask(with: command.urlRequest) { (_, response, error) in
            if /* success */ {
                success?()
            } else /* error */ {
                failure?(error)
            }
        }

        task.resume()
    }
    func execute<Q: Query, Result>(_ query: Q,
                                   success: QuerySuccessCallback<Result>?,
                                   failure: FailureCallback?) where Result == Q.Result {
        let task = session.dataTask(with: query.urlRequest) { [jsonDecoder] (data, _, error) in
            if /* error */ {
                failure?(error)
            } else if let data = data {
                let result = try! jsonDecoder.decode(Result.self, from: data)
                success?(result)
            } else {
                // additional error handling ...
            }
        }

        task.resume()
    }
}
```

Specialized command protocol example

```
protocol IdCommand: Command {
    var id: String { get }
}

extension IdCommand {
    var bodyDict: JSON {
        return ["id": id]
    }
}
```

Usage - command for liking videos

```swift
struct LikeVideoCommand: IdCommand {

    static let path = "like_video"
    static let method = .post

    let id: String
}

apiClient.execute(
    LikeVideoCommand(id: "let_swift_13_speaker_1_intro"),
    success: { print("👏 🎉 👍") },
    failure: { print("😢") }
)
```

Specialized command protocol example

```swift
protocol CommandBody {
    var json: JSON { get }
}

protocol CommandWithBody: Command {
    associatedType Body: CommandBody

    var body: Body
}

extension CommandWithBody {
    var bodyDict: JSON {
        return body.json
    }
}
```

Usage - command for adding new speaker

```
struct AddSpeakerCommandBody: CommandBody {

    let id: String
    let firstName: String
    let lastName: String

    var json: JSON {
        // create json from fields above
        // or just make it Encodable and tell compiler to do the dirty job
        ...
    }
}
```

Usage - command for adding new speaker

```
struct AddSpeakerCommand: CommandWithBody {

    static let path = "add_speaker"
    static let method = .post

    let body: AddSpeakerCommandBody
}
```

Specialized query protocols

```
/// Protocol for queries which return one entity with given id
protocol IdQuery {
    var id: String
}

/// Protocol for queries which should return results in specific order.
protocol Orderable {
    var order: OrderType { get }
}

/// Protocol for queries which are pageable.
protocol Pageable {
    var page: String? { get }
}
```

Specialized query protocols

```swift
/// Protocol which identifies queries which can return results based on string query
protocol Searchable {
    var query: String? { get }
}

/// Protocol for queries which return items filtered by given parameters
protocol Filterable {
    associatedtype Filter: FilterProtocol

    var filter: Filter? { get }
}

protocol FilterProtocol {
    var filtersDict: [String: String] { get }
}
```

```
extension Query where Self: Filterable {
    var parameters: [String: String]? {
        return filter?.filtersDict
    }
}
```

```
extension Query where Self: Filterable & Pageable {
    var parameters: [String: String]? {
        var dictionary = filter?.filtersDict ?? [:]
        if let page = page {
            dictionary["page"] = page
        }
        return dictionary
    }
}
```

```swift
extension Query where Self: Filterable & Orderable & Pageable {
    var parameters: [String: String]? {
        var dictionary = filter?.filtersDict ?? [:]
        dictionary["order"] = order.rawValue

        if let cursor = cursor {
            dictionary["page"] = page
        }
        return dictionary
    }
}
```

Example usage

```swift
struct SpeakersQuery: Query, Filterable, Pageable {
    typealias Result = [Speaker]

    static let path = "speakers"

    let filter: SpeakerFilter?
    let page: String?
}
```

```
struct SpeakerFilter: FilterProtocol {
    let name: String?
    let numberOfLetSwiftsAttended: Int?

    var filtersDict: [String: String] {
        // create dict from fields
        ...
    }
}
```

```swift
let filter = SpeakerFilter(
    name: "Sebastian",
    numberOfLetSwiftsAttended: 13
)

apiClient.fetch(
    SpeakersQuery(filter: filter, page: "start"),
    success: { speakers in print(speakers) },
    failure: nil
)
```

Cons

- Lack of namespacing.
  Solution: nested structs

- It's possible to forget to implement extension for given combination of protocols and compiler won't warn us.
  Solution: unit tests

- Some code repetition in extensions.
  Solution: extracting building body/parameters dictionaries to static methods or builder

Pros

- All request's configuration in one place - without reading multiple switches
- It's easy to model similar requests and add new specialized requests
- Queries can define their return types

What we gained?

- Better way to manage all commands and queries
- Removed some dirty hacks
- [UNEXPECTED] We were able to extract whole API layer to framework and reused it in other project for the same client

THANK YOU!

This presentation can be found here:

https://github.com/SebastianOsinski/LetSwiftSlides

QUESTIONS?