from problem to solution

soroush khanlou

pragma conf 2019

@khanlou

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

this is testable

how do we know it’s testable?

only one singleton

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session: URLSessionProtocol = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session: URLSessionProtocol

    init(session: URLSessionProtocol = URLSession.shared) {
        self.session = session
    }

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

cyclomatic complexity

cyclomatic complexity is branches

how many branches do we have?

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

1. happy path
2. network error
3. status code error
4. json decoding error

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

good code never
gets to stay good

what happens when we try to add a feature?

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
               return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
```

```
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
               return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
               return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })
```

⚠ 'isNetworkActivityIndicatorVis

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })
```

```
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

what happened??

we need to simplify this

why now?

single responsibility principle

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

cyclomatic complexity

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}
```

2

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest
```

2

```
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }
```

3

```
        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2

2

3

4

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2 2 3 4 2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

2

×2

×3

×4

×2

2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

× 2

× 3

× 4

× 2

=

96

2

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}
```

× 2

× 3

× 4

× 2

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

=

96 tests if we want to be complete!

line length

```swift
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        let urlRequest = URLRequestBuilder(request: request).urlRequest

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
    }
}
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

so what can we do?

deduplicate?

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }

    func expire(_ identifier: UIBackgroundTaskIdentifier?) {
        if let identifier = identifier {
            UIApplication.shared.endBackgroundTask(identifier)
        }
    }
}
```

```swift
class RequestCounter {

    static let shared = RequestCounter()

    var counter = 0 {
        didSet {
            UIApplication.shared.isNetworkActivityIndicatorVisible = counter == 0
        }
    }
}

final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {

            self.expire(identifier)

        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ (data, response) -> Output in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({

                expire(identifier)

                RequestCounter.shared.counter -= 1
            })
    }

    func expire(_ identifier: UIBackgroundTaskIdentifier?) {
        if let identifier = identifier {
            UIApplication.shared.endBackgroundTask(identifier)
        }
    }
}
```

deduplication

can’t help us here because we don’t have a lot of code that is repeated

extract functions?

```swift
RequestCounter.shared.counter += 1

// ...

RequestCounter.shared.counter -= 1
```

```swift
func increment() {
    RequestCounter.shared.counter += 1
}

func decrement() {
    RequestCounter.shared.counter -= 1
}

increment()

decrement()
```

```swift
func adjust(by value: Int) {
    RequestCounter.shared.counter += value
}

func increment() {
    RequestCounter.shared.counter += 1
}

func decrement() {
    RequestCounter.shared.counter -= 1
}

increment()

decrement()
```

```swift
func adjust(by value: Int) {
    RequestCounter.shared.counter += value
}

func increment() {
    adjust(by: 1)
}

func decrement() {
    adjust(by: -1)
}

increment()

decrement()
```

```swift
// Constants.swift
let kNetworkingAdjustmentIncrement = 1
let kNetworkingAdjustmentDecrement = -1

// NetworkClient.swift
func adjust(by value: Int) {
    RequestCounter.shared.counter += value
}

func increment() {
    adjust(by: kNetworkingAdjustmentIncrement)
}

func decrement() {
    adjust(by: kNetworkingAdjustmentDecrement)
}

increment()

decrement()
```

extraction

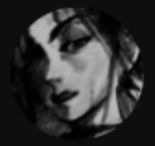
doesn’t help here because we’re not
adding any new knowledge or ability,
just moving things around

```swift
func adjust(by value: Int) {
    RequestCounter.shared.counter += value
}

func increment() {
    adjust(by: 1)
}

func decrement() {
    adjust(by: -1)
}

increment()

decrement()
```

Anathema @BitterAmethyst · Dec 4

I want to beat OOP programmers over the head with a giant sign that says "**Indirection** is not **Abstraction**"

1 3

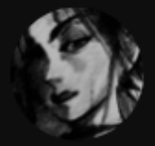


Anathema @BitterAmethyst · Dec 4

I want to beat OOP programmers over the head with a giant sign that says "**Indirection** is not **Abstraction**"

Sam Buchanan @afongen · 10 Jul 2007

Digging through unpleasant source code to fix a bug. Repeat after me: **indirection** is not **abstraction**.



Anathema @BitterAmethyst · Dec 4

I want to beat OOP programmers over the head with a giant sign that says "**Indirection** is not **Abstraction**"

Sam Buchanan @afongen · 10 Jul 2007

Digging through unpleasant source code to fix a bug. Repeat after me: **indirection** is not **abstraction**.

Gabriel Claramunt @gclaramunt · 9 May 2016

Please, don't confuse **abstraction** with **indirection**

4 4

Anathema @BitterAmethyst · Dec 4

I want to beat OOP programmers over the head with a giant sign that says "**Indirection** is not **Abstraction**"

Sam Buchanan @afongen · 10 Jul 2007

Digging through unpleasant source code to fix a bug. Repeat after me: **indirection** is not **abstraction**.

Gabriel Claramunt @gclaramunt · 9 May 2016

Please, don't confuse **abstraction** with **indirection**

Jessica Kerr @jessitron · 23 Sep 2014

OH: We love **indirection**! It's kind of like **abstraction**, only easier!

1 9 7

Anathema @BitterAmethyst · Dec 4

I want to beat OOP programmers over the head with a giant sign that says "**Indirection** is not **Abstraction**"

Sam Buchanan @afongen · 10 Jul 2007

Digging through unpleasant source code to fix a bug. Repeat after me: **indirection** is not **abstraction**.

Gabriel Claramunt @gclaramunt · 9 May 2016

Please, don't confuse **abstraction** with **indirection**

Jessica Kerr @jessitron · 23 Sep 2014

OH: We love **indirection**! It's kind of like **abstraction**, only easier!

Gary Bernhardt @garybernhardt · 22 Dec 2014

Before using the word "**abstraction**", or even forming an opinion about it, explain the difference between it and **indirection** to someone.

3 24 31

we're here to learn

how to abstract

step 1 - find the similarities in the problem

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            urlRequest.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest


        authToken.add(to: urlRequest)


        var identifier: UIBackgroundTaskIdentifier?
        identifier = UIApplication.shared.beginBackgroundTask(expirationHandler: {
            if let identifier = identifier {
                UIApplication.shared.endBackgroundTask(identifier)
            }
        })

        RequestCounter.shared.counter += 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {
    let session = URLSession.shared
    let authToken = AuthToken()
    let counter = Counter()
    let backgrounding = Backgrounding()
    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {
        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)



        backgrounding.prepare()



        RequestCounter.shared.counter -= 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                if let identifier = identifier {
                    UIApplication.shared.endBackgroundTask(identifier)
                }
                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {
    let session = URLSession.shared
    let authToken = AuthToken()
    let counter = Counter()
    let backgrounding = Backgrounding()
    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {
        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)



        backgrounding.prepare()



        RequestCounter.shared.counter -= 1

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({

                backgrounding.release()

                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {
    let session = URLSession.shared
    let authToken = AuthToken()
    let counter = Counter()
    let backgrounding = Backgrounding()
    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {
        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)



        backgrounding.prepare()



        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({

                backgrounding.release()

                RequestCounter.shared.counter -= 1
            })
    }
}
```

auth token - backgrounding - counter

by responsibility

```
final class NetworkClient {
    let session = URLSession.shared
    let authToken = AuthToken()
    let counter = Counter()
    let backgrounding = Backgrounding()
    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {
        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)



        backgrounding.prepare()



        counter.increment()
        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

auth token - backgrounding - counter

```swift
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

before

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.decrement()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

headers

before

after

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

```swift
protocol RequestBehavior {
    func addHeaders(to request: URLRequest)
    func before()
    func after()
}
```

```
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

why is this bit of error handling special?

```swift
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.addHeaders(to: urlRequest)
        backgrounding.before()
        counter.before()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.after()
                counter.after()
            })
    }
}
```

why treat these two
things differently?

“a good idea is something that does not solve just one single problem, but rather can solve multiple problems at once.” – shigeru miyamoto

my epiphany

Cocoaphony

Blog | Archives | RSS

22 APR 2019

Protocols I: "Start With a Protocol," He Said

In the beginning, Crusty

In 2015, at WWDC, Dave Abrahams gave what I believe is still the greatest Swift talk ever given, and certainly the most influential. ”Protocol-Oriented Programming in Swift,” or as it is more affectionately known, “The Crusty Talk.”

This is the talk that introduced the phrase “protocol oriented programming.” The first time I watched it, I took away just one key phrase:

> *Start with a protocol.*

`AuthToken`

`Backgrounding`

`URLSession`

how can we treat all these different things the same?

`StatusCode`

`Counter`

```swift
final class NetworkClient {

    let session = URLSession.shared

    let authToken = AuthToken()

    let counter = Counter()

    let backgrounding = Backgrounding()

    func send<Output: Codable>(request: Request<Output>) -> Promise<Output> {

        var urlRequest = URLRequestBuilder(request: request).urlRequest

        authToken.add(to: urlRequest)
        backgrounding.prepare()
        counter.increment()

        return session.data(with: urlRequest)
            .then({ data, response in
                guard (200..<300).contains(response.statusCode) else {
                    throw StatusCodeError(statusCode: response.statusCode)
                }
                return try JSONDecoder().decode(Output.self, from: data)
            })
            .always({
                backgrounding.release()
                counter.decrement()
            })
    }
}
```

```
authToken.add(to: urlRequest)
backgrounding.prepare()
counter.increment()

       session.data(with: urlRequest)

        backgrounding.release()
        counter.decrement()
```

```
authToken.add(to: urlRequest)
backgrounding.prepare()
counter.increment()

       session.data(with: urlRequest)

        counter.decrement()
        backgrounding.release()
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()

            session.data(with: urlRequest)





        counter.decrement()
    backgrounding.release()
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()

            session.data(with: urlRequest)





        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()
           session.data(with: urlRequest)
        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()
            session.data(with: urlRequest)
        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()
            // no action
                session.data(with: urlRequest)
            statusCodes.validate()
        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()
            // no action
                URLSession.shared
            statusCodes.validate()
        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        counter.increment()
            StatusCodeValidating(wrapping:
                URLSession.shared
            )
        counter.decrement()
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    backgrounding.prepare()
        RequestCounting(wrapping:
            StatusCodeValidating(wrapping:
                URLSession.shared
            )
        )
    backgrounding.release()
// no action
```

```
authToken.add(to: urlRequest)
    Backgrounding(wrapping:
        RequestCounting(wrapping:
            StatusCodeValidating(wrapping:
                URLSession.shared
            )
        )
    )
// no action
```

```
Authenticating(wrapping:
    Backgrounding(wrapping:
        RequestCounting(wrapping:
            StatusCodeValidating(wrapping:
                URLSession.shared
            )
        )
    )
)
```

```
Authenticating(wrapping:
    Backgrounding(wrapping:
        RequestCounting(wrapping:

                URLSession.shared
                    .checkingStatusCodes()
        )
    )
)
```

```
Authenticating(wrapping:
    Backgrounding(wrapping:

                URLSession.shared
                    .checkingStatusCodes()
                    .countingRequests()
    )
)
```

```
Authenticating(wrapping:

                    URLSession.shared
                        .checkingStatusCodes()
                        .countingRequests()
                        .handlingBackgroundTasks()
)
```

```
URLSession.shared
    .checkingStatusCodes()
    .countingRequests()
    .handlingBackgroundTasks()
    .authenticating()
```

```
URLSession.shared
    .checkingStatusCodes()
    .countingRequests()
    .handlingBackgroundTasks()
    .authenticating()
```

step 1 - find the similarities in the problem

step 1 - find the similarities in the problem

step 2 - develop the abstraction

```
protocol Transport {
    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)>
}
```

```swift
extension URLSession: Transport {
    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        // ...
    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }





}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {




    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {

        return self.wrapped.send(request: request).then({ data, response in


        })
    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        //before send
        return self.wrapped.send(request: request).then({ data, response in
            //after success

        })
    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return self.wrapped.send(request: request).then({ data, response in



        })
    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return self.wrapped.send(request: request).then({ data, response in
            guard (200..<300).contains(response.statusCode) else {

            }
        })
    }
}
```

```swift
final class StatusCodeCheckingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return self.wrapped.send(request: request).then({ data, response in
            guard (200..<300).contains(response.statusCode) else {
                throw StatusCodeError(statusCode: response.statusCode)
            }
        })
    }
}
```

```
StatusCodeCheckingTransport(wrapping:
    URLSession.shared
)
```

```
URLSession.shared
    .checkingStatusCodes()
```

```
extension Transport {

}
```

```swift
extension Transport {
    func checkingStatusCodes() -> Transport {

    }
}
```

```swift
extension Transport {
    func checkingStatusCodes() -> Transport {
        return StatusCodeCheckingTransport(wrapping: self)
    }
}
```

```
URLSession.shared
    .checkingStatusCodes()
```

```swift
final class HeaderAddingTransport: Transport {

}
```

```swift
final class HeaderAddingTransport: Transport {

    let wrapped: Transport
    let headers: [String: String]

    init(wrapping: Transport, headers: [String: String]) {
        self.wrapped = wrapping
        self.headers = headers
    }





}
```

```swift
final class HeaderAddingTransport: Transport {

    let wrapped: Transport
    let headers: [String: String]

    init(wrapping: Transport, headers: [String: String]) {
        self.wrapped = wrapping
        self.headers = headers
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {





    }
}
```

```swift
final class HeaderAddingTransport: Transport {

    let wrapped: Transport
    let headers: [String: String]

    init(wrapping: Transport, headers: [String: String]) {
        self.wrapped = wrapping
        self.headers = headers
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request




    }
}
```

```swift
final class HeaderAddingTransport: Transport {

    let wrapped: Transport
    let headers: [String: String]

    init(wrapping: Transport, headers: [String: String]) {
        self.wrapped = wrapping
        self.headers = headers
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        for (key, value) in headers {
            mutableCopy.addValue(value, forHTTPHeaderField: key)
        }

    }
}
```

```swift
final class HeaderAddingTransport: Transport {

    let wrapped: Transport
    let headers: [String: String]

    init(wrapping: Transport, headers: [String: String]) {
        self.wrapped = wrapping
        self.headers = headers
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        for (key, value) in headers {
            mutableCopy.addValue(value, forHTTPHeaderField: key)
        }
        return self.wrapped.send(request: mutableCopy)
    }
}
```

```swift
let transport: Transport =
    HeaderAddingTransport(
        wrapping: URLSession.shared,
        headers: [
            "Content-Type": "application/json",
            "Accept": "application/json",
        ]
    )
```

```swift
extension Transport {
    func addingJSONHeaders() -> Transport {
        return HeaderAddingTransport(
        wrapping: self,
        headers: [
            "Content-Type": "application/json",
            "Accept": "application/json",
        ]
    }
}
```

```
URLSession.shared
    .checkingStatusCodes()
    .addingJSONHeaders()
```

step 1 - find the similarities in the problem

step 2 - develop the abstraction

a word on “don’t repeat yourself”

	similar code	dissimilar code
similar underlying concepts		
dissimilar underlying concepts		

	similar code	dissimilar code
similar underlying concepts		
dissimilar underlying concepts		most code

	similar code	dissimilar code
similar underlying concepts	probably already “abstracted”	
dissimilar underlying concepts		most code

	similar code	dissimilar code
similar underlying concepts	probably already "abstracted"	
dissimilar underlying concepts	**a world of pain (false positive)**	most code

	similar code	**dissimilar code**
similar underlying concepts	probably already "abstracted"	**ripe for abstraction (false negative)**
dissimilar underlying concepts	a world of pain (false positive)	most code

“duplication is far cheaper than the wrong abstraction” - sandi metz

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

```swift
final class MockTransport: Transport {
    let data: Data
    let response: HTTPURLResponse

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return Promise(value: (data, response))
    }

}
```

```swift
final class MockTransport: Transport {
    let data: Data
    let response: HTTPURLResponse

    init(data: Data, response: HTTPURLResponse) {
        self.data = data
        self.response = response
    }









    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return Promise(value: (data, response))
    }



}
```

```swift
final class MockTransport: Transport {
    let data: Data
    let response: HTTPURLResponse

    init(data: Data, response: HTTPURLResponse) {
        self.data = data
        self.response = response
    }

    init(statusCode: Int, data: Data = Data()) {





    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return Promise(value: (data, response))
    }



}
```

```swift
final class MockTransport: Transport {
    let data: Data
    let response: HTTPURLResponse

    init(data: Data, response: HTTPURLResponse) {
        self.data = data
        self.response = response
    }

    init(statusCode: Int, data: Data = Data()) {
        self.data = data
        self.response = HTTPURLResponse(
            url: URL(string: "example.com")!,
            statusCode: statusCode,
            httpVersion: nil,
            headerFields: nil
        )!
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return Promise(value: (data, response))
    }



}
```

```swift
final class MockTransport: Transport {
    let data: Data
    let response: HTTPURLResponse

    init(data: Data, response: HTTPURLResponse) {
        self.data = data
        self.response = response
    }

    init(statusCode: Int, data: Data = Data()) {
        self.data = data
        self.response = HTTPURLResponse(
            url: URL(string: "example.com")!,
            statusCode: statusCode,
            httpVersion: nil,
            headerFields: nil
        )!
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        return Promise(value: (data, response))
    }

    static let ok: Transport = MockTransport(statusCode: 200)
    static let serverError: Transport = MockTransport(statusCode: 500)
}
```

```
let viewController = MyViewController(transport: MockTransport.serverError)
```

```swift
final class RequestInspectableTransport: Transport {

}
```

```swift
final class RequestInspectableTransport: Transport {
    var lastSeenRequest: URLRequest?

}
```

```swift
final class RequestInspectableTransport: Transport {
    var lastSeenRequest: URLRequest?

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }




}
```

```swift
final class RequestInspectableTransport: Transport {
    var lastSeenRequest: URLRequest?

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {


    }
}
```

```swift
final class RequestInspectableTransport: Transport {
    var lastSeenRequest: URLRequest?

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        lastSeenRequest = request

    }
}
```

```swift
final class RequestInspectableTransport: Transport {
    var lastSeenRequest: URLRequest?

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        lastSeenRequest = request
        return self.wrapping.send(request: request)
    }
}
```

```
transport.send(request: request)
    .then({ data, response in
    })
```

```
transport.send(request: request)

    .then({ data, response in
        // but I don't want Data, I want my model object
    })
```

```
transport.send(request: request)
    .then({ data, response in
    })
```

```
transport.send(request: request)
    .decode(User.self)
    .then({ data, response in

    })
```

```
transport.send(request: request)
    .decode(User.self)
    .then({ user in

    })
```

```
transport.send(request: request)
    .decode(User.self)
    .then({ user in
        // yay
    })
```

```swift
extension Promise                                    {



}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {



}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode                          (_ type:                    ) -> Promise<            > {



    }
}
```

```
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode          >(_ type:                 ) -> Promise<           > {



    }
}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode          >(_ type: TypeToDecode.Type) -> Promise<           > {



    }
}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode          >(_ type: TypeToDecode.Type) -> Promise<TypeToDecode> {



    }
}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode: Decodable>(_ type: TypeToDecode.Type) -> Promise<TypeToDecode> {



    }
}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode: Decodable>(_ type: TypeToDecode.Type) -> Promise<TypeToDecode> {
        return self.then({ data, response in

        })
    }
}
```

```swift
extension Promise where Value == (Data, HTTPURLResponse) {
    func decode<TypeToDecode: Decodable>(_ type: TypeToDecode.Type) -> Promise<TypeToDecode> {
        return self.then({ data, response in
            return try JSONDecoder().decode(type, from: data)
        })
    }
}
```

```
transport.send(request: request)
    .decode(User.self)
    .then({ user in
        // yay
    })
```

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

step 4 - extract

```swift
final class AuthenticatingTransport: Transport {

}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }





}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {





    }
}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request




    }
}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {

        }

    }
}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            mutableCopy.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }

    }
}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            mutableCopy.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }
        return self.wrapped.send(request: mutableCopy)
    }
}
```

```swift
extension Transport {
    func authenticating() -> Transport {
        return AuthenticatingTransport(wrapping: self)
    }
}
```

```
URLSession.shared
    .addingJSONHeaders()
    .checkingStatusCodes()
    .authenticating()
```

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

step 4 - extract

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

step 4 - extract

step 5 - reap the benefits

what else can we build?

```
final class TimingTransport: Transport {

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }









}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {





    }

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {

        return self.wrapping.send(request: request)




    }

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        let startDate = Date()
        return self.wrapping.send(request: request)




    }

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        let startDate = Date()
        return self.wrapping.send(request: request)
            .always({


            })
    }

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        let startDate = Date()
        return self.wrapping.send(request: request)
            .always({
                let delta = -startDate.timeIntervalSinceNow

            })
    }

}
```

```swift
final class TimingTransport: Transport {

    let wrapping: Transport

    init(wrapping: Transport) {
        self.wrapping = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        let startDate = Date()
        return self.wrapping.send(request: request)
            .always({
                let delta = -startDate.timeIntervalSinceNow
                print("request took \(delta) seconds")
            })
    }

}
```

step 1 - find the similarities in the problem

step 2 - develop the abstraction

step 3 - build commonalities and tools

step 4 - extract

step 5 - reap the benefits

so what did we gain?

decoupling

decoupling is important because it allows you to **build strong boundaries**, which allows you to **handle separate components separately**

what do i mean
by “handle”?

you can test it

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            mutableCopy.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }
        return self.wrapped.send(request: mutableCopy)
    }
}
```

```swift
final class AuthenticatingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        var mutableCopy = request
        if let authToken = UserDefaults.standard.string(forKey: "authToken") {
            mutableCopy.addValue(authToken, forHTTPHeaderField: "X-Auth-Token")
        }
        return self.wrapped.send(request: mutableCopy)
    }
}
```

you can reuse it

pre-authentication

```
URLSession.shared
    .addingJSONHeaders()
    .checkingStatusCodes()
```

post-authentication

```
URLSession.shared
    .addingJSONHeaders()
    .checkingStatusCodes()
    .authenticating()
```

you can glance at it

```swift
final class LoggingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        print("sending request \(request.url)")
        return self.wrapped.send(request: request).then({ data, response in
            print("status code \(response.statusCode)")
        })
    }

}
```

```swift
final class LoggingTransport: Transport {

    let wrapped: Transport

    init(wrapping: Transport) {
        self.wrapped = wrapping
    }

    func send(request: URLRequest) -> Promise<(Data, HTTPURLResponse)> {
        print("sending request \(request.url)")
        return self.wrapped.send(request: request).then({ data, response in
            print("status code \(response.statusCode)")
        })
    }

}
```

“there is no abstract art. you must always start with something. afterward you can remove all traces of reality.” – pablo picasso