Farm to Table

Nordstrom - Customer Mobile Applications

Who are we?

We are an integrated product team
Developers
Designers
QA
Program Managers
Product Managers
People Managers

15-20 iOS Developers

Pair Programming
Test Driven Development

We make the iOS Customer Mobile App!

iOS Customer Mobile App

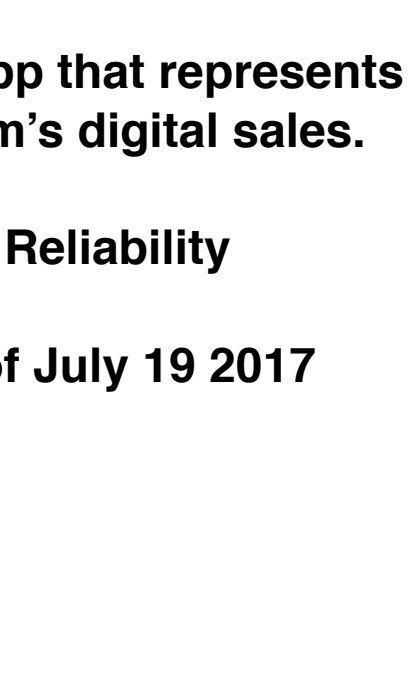
We are merchants at heart.

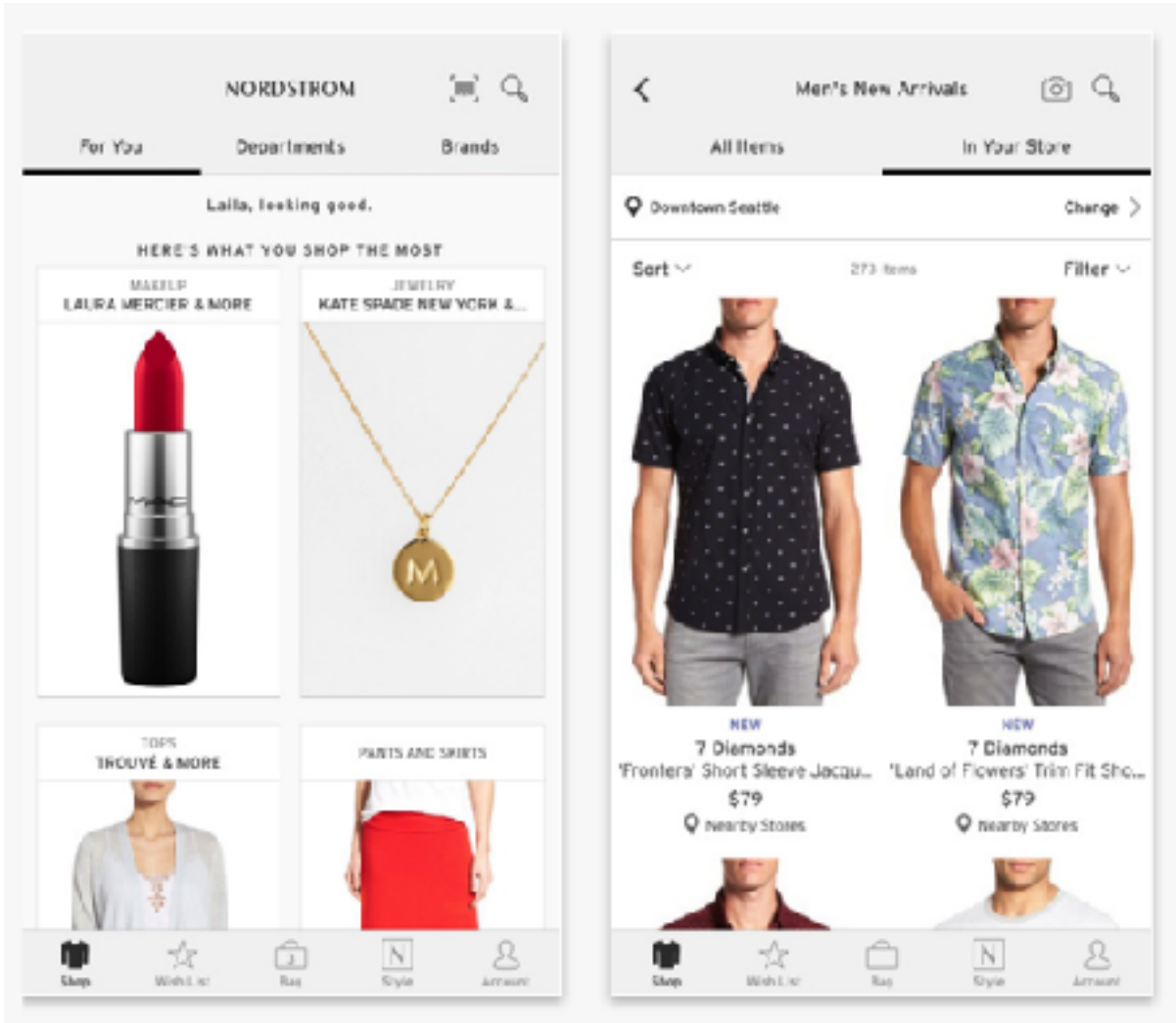
The iOS mobile app is a growing retail channel.

We have a great app that represents a solid share of Nordstrom's digital sales.

We have a 99.99% Reliability

iTunes Rating as of July 19 2017





NORDSTROM TECHNOLOGY

Challenges: Aggressive Feature Growth

We had trouble scaling the engineering team to beyond 20 developers

It was difficult to bring new engineers up to speed fast enough

It was difficult trying to let other teams build off of our platform

It was difficult to carve out pieces of the app to hand to other teams

Possible Drivers

Inherited consultant codebase designed 5 years ago

Heavy Bias for Delivery

Too much variation in how code was implemented

Not enough separation of concerns

New Engineers didn’t have easy rules to follow

SWIFT REWRITE - NEW Architectural Goals

Easy to understand what level of abstraction to use and where

We should be able to scale horizontally

Testing needs to be easy. Input/Output

Infrastructure should help enforce good separation of concerns

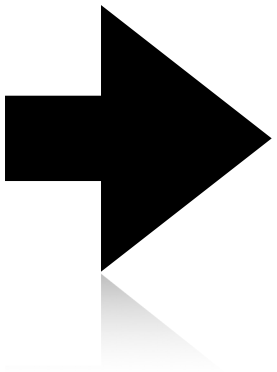
Should be viewed more as a platform than a monolithic app

Data Modeling

The right level of data modeling for each layer

Perhaps analogy is the best way to articulate this?

Enter FARM To TABLE



FARM to TABLE: Engineers drawing on whiteboards

COW

RAW MILK

ICECREAM FACTORY

ICECREAM

ICE CREAM PARLOR (KITCHEN)

PLATED ICECREAM

WE WANT ICE CREAM, NOT MILK OR COWS

CREATE A SERVICE ABSTRACTION LAYER

We created a separate module that abstracts all the local backend service calls

This layer's job is to provide the App with the best domain model of the data

Additional composition can be done in this layer if needed

We ended up calling this module NBApi

NBAPI VENDS THE APP'S IDEAL VERSION OF THE DOMAIN MODEL

Current state of iOS services

Future state of iOS services

Domain Service

THE KITCHEN - WHERE DOMAIN MODELS INTERSECT

The kitchen can transform domain models into rich state models the views can use to render themselves

Input: Immutable VCActions (user interaction, initial loading, etc)

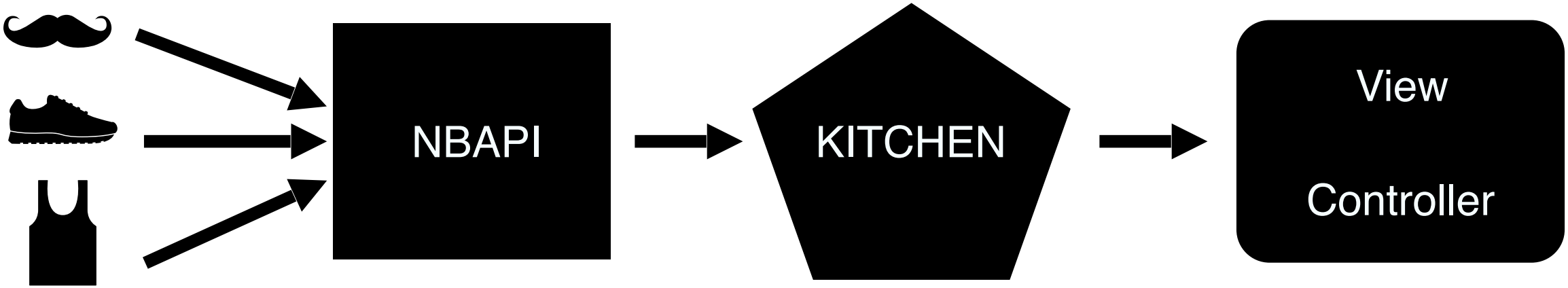
Output: Immutable VCState model events (model data used to update UI)

THE KITCHEN - WHERE DOMAIN MODELS INTERSECT

The Kitchen cooks up various domain models and vends view friendly models

Inspired by Facebook's Flux, Redux, Elm, Event Sourcing, and Jake Wharton's Managing State in RX presentation

SERVICES TO VIEW DATA FLOW



Kitchen Code Example

Input: Observable<LoginViewAction>
Output: Observable<LoginViewState>

```
func bindTo(events: Observable<LoginViewAction>) -> Observable<LoginViewState> {

    return events.flatMap({ [unowned self] (loginViewAction) -> Observable<LoginViewState> in
        switch loginViewAction {
        case .textChanged(let textType):
            return self.validateTextType(textType)
        case .submitButtonPressed(let usernameTextType, let passwordTextType):
            return self.validateOnSubmit(usernameTextType, passwordTextType)
                    .startsWith(.startedLoading)
                    .concat(Observable.just(.finishedLoading)
        }
    })

}
```

Kitchen Test Example

```
beforeEach {
    outputEvents = [LoginViewState]()
    fakeFieldValidating = FakeFieldValidating()
    fakeCoronaService = FakeCoronaService()
    inputActionSource = PublishSubject<LoginViewAction>()
    subject = AuthenticationKitchen(fieldValidating: fakeFieldValidating, coronaService: fakeCoronaService)

    subject.bindTo(actions: inputActionSource.asObservable()).subscribe(onNext: { (outputViewState) in
        outputEvents.append(outputViewState)
    }).disposed(by: disposeBag)
}

context("when we receive the submitButtonPressed action") {
    context("when the fields pass validation") {
        beforeEach {
            fakeFieldValidating.stubbedUsernameResult = true
            fakeFieldValidating.stubbedPasswordResult = true
            fakeCoronaService.stubbedReturn = "FakeCorona"
            inputActionSource.onNext(.submitButtonPressed(.username("goodstuff"), .password("moregoodstuff")))
        }

        it("should have output a startedLoading, followed by a success() state followed by a finishedLoading state") {
            let expectedOutput: [LoginViewState] = [.startedLoading, .success("FakeCorona"), .finishedLoading]
            let isValid = expectedOutput == outputEvents
            expect(isValid).to(beTrue())
        }
    }
}
```

For a small project example of Farm to Table

https://github.com/khappucino/2Farm2Furious

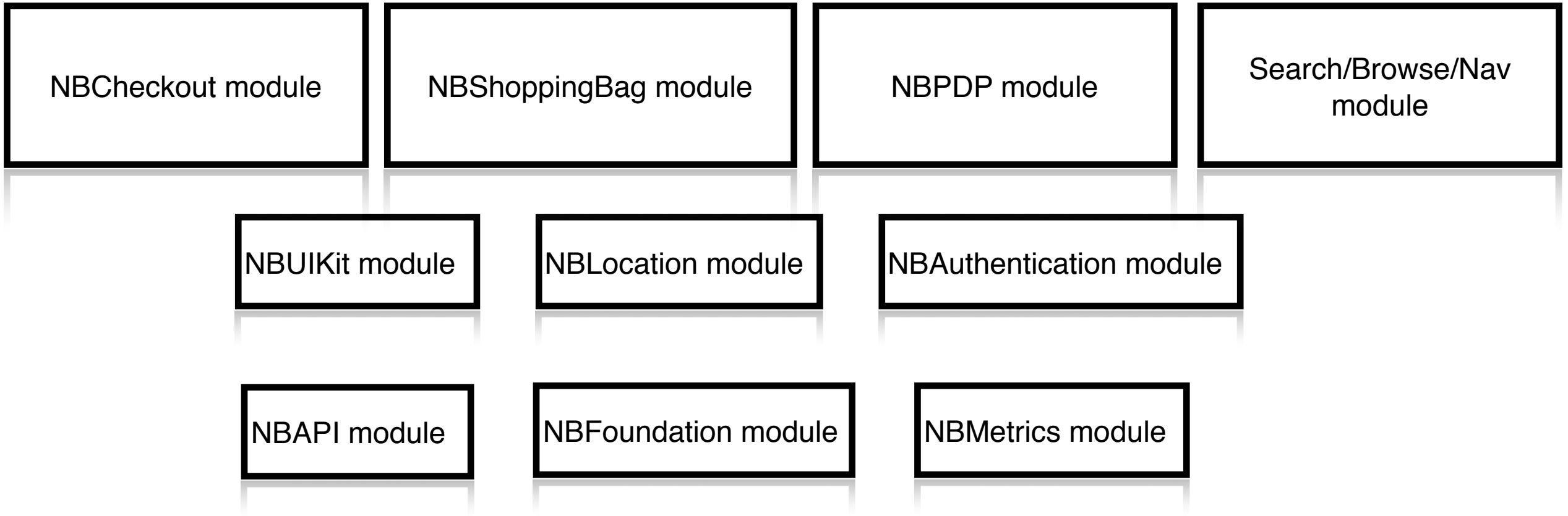
Modules

Enforcing access controls

Reminding developers to think if they should be accessing items within a context

App Modularization

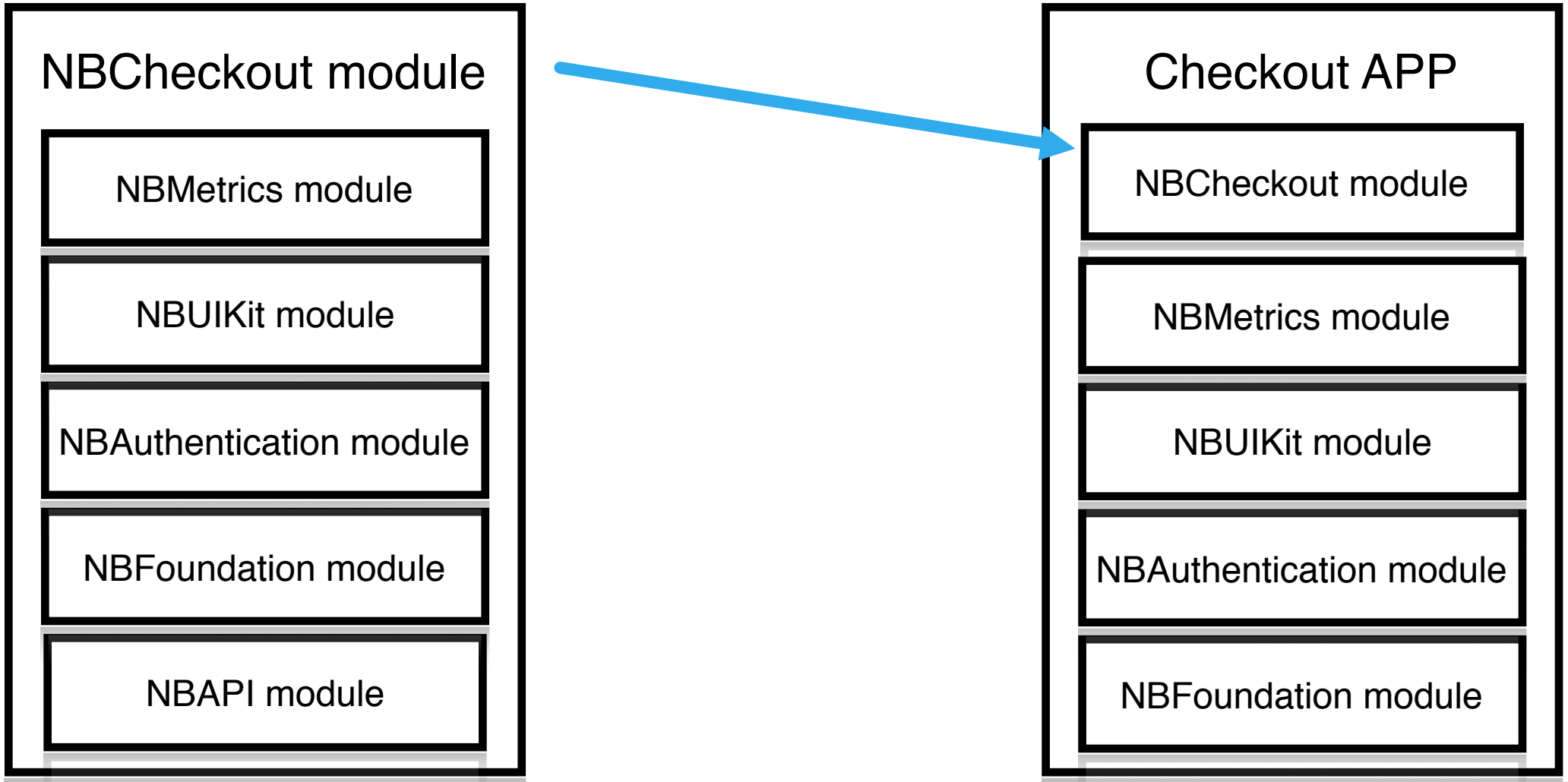
NBApi is in a separate git repo



CHECKOUT Module example

NBCheckout module handles it's own internal routes.
Only the initial ViewController is public

The Checkout app is a stand alone app.
It allows you to create an app of just the checkout flow



Modules

- Create a .podspec for each module
- The .podspec should point to the source (which is in the same repo)
- Add pods as usual