Way to reactivity

- The development of Android architectures
- RxJava
- How to master reactivity with MVI
I am here to tell you something about MVI
Android Architectures

The early days

In the beginning, there was silence...
— Daniel Barenboim
Android Architectures

Nowadays

...And out of the silence came the sound.
— Daniel Barenboim
Android Architectures Nowadays

- Separation of concerns
  - Expandable
  - Maintainable
  - Testable
- Passive views
  - All actions through the presenter / view model
RxJava
RxJava

Observable

  .just("Hello world")
  .doOnNext { println(it) }
  .debounce(1, TimeUnit.SECONDS)
  .subscribe { println(it) }
RxAndroid
RxAndroid

Observable

   .from(listOf("one", "two", "three", "four", "five"))
   .subscribeOn(Schedulers.computation())
   .observeOn(AndroidSchedulers.mainThread())
   .subscribe(/* an Observer */);
RxAndroidLibs

- RxActivityResult - A reactive-tiny-badass-vindictive library to break with the OnActivityResult implementation as it breaks the observables chain.
- RxAnimations - Repository for android animations Rx wrapper
- RxBinding - RxJava binding APIs for Android's UI widgets
- RxLifecycle - Lifecycle handling APIs for Android apps using RxJava
- RxPaparazzo - RxJava extension for Android to take images using camera and gallery
- RxRecyclerView - Reactive RecyclerView Adapter

Data
- ReactiveCache - A reactive cache for Android and Java which honors the Observable chain.
- RxPreferences - Reactive SharedPreferences for Android
- RxCache - Reactive caching library for Android and Java
- RxCuzboard - Store and retrieve streams of POJOs from an Android database using RxJava and Cupboard
- RxFileObserver - Simple reactive API for Android's FileObserver class
- RxFileWatcher - Convenient file watcher with an RxJava interface, based on JDK WatchService
- RxLoader - An Android Loader that wraps an RxJava Observable.
- RxStore - A tiny library that assists in saving and restoring objects to and from disk using RxJava.
- SQLite - A lightweight wrapper around SQLiteOpenHelper
- StorIo - Beautiful API for SQLiteDatabase and ContentResolver

Network
- ReactiveNetwork - Android library listening network connection state and change of the WiFi signal strength with RxJava Observables
- Retrofit - Type-safe REST client for Android and Java
- RxBonjour - Reactive network service discovery
- Wasp - Compact and easy to use, 'all-in-one' android network solution

Testing
- RxMocks - Mocks/Assertions for RxJava testing
The Data Flow Pitfall - 1

RxView
    .clicks(submitView)
    .doOnNext {
        submitView.setEnabled(false);
        progressView.setVisibility(VISIBLE);
    }
    .flatMap { service.setName(nameView.text.toString()) }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(/* an Observer */)
The Data Flow Pitfall - 2

RxView

  .clicks(submitView)
  .doOnNext {
    submitView.setEnabled(false);
    progressView.setVisibility(VISIBLE);
  }
  .flatMap { service.setName(nameView.text.toString()) }
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(/* an Observer */)
The Data Flow Pitfall - 3

RxView

  .clicks(submitView)
  .doOnNext {
    submitView.setEnabled(false);
    progressView.setVisibility(VISIBLE);
  }
  .flatMap { service.setName(nameView.text.toString()) }
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(/* an Observer */)
The Data Flow Pitfall - 4

RxView

|.clicks(submitView)
|.doOnNext {
    submitView.setEnabled(false);
    progressView.setVisibility(VISIBLE);
}
|.flatMap { service.setName(nameView.text.toString()) }  
|.observeOn(AndroidSchedulers.mainThread())  
|.subscribe(/* an Observer */)
The Data Flow Pitfall - 5
The Data Flow Pitfall - 7
The State Problem
After one moment of inattention
Model View Intent

- Originally defined by André Staltz for cycle.js
- Transferred from cycle.js to Android by Hannes Dorfmann
Model

- Single source of truth for...
  - ...business Logic
  - ...and view
- Model reflects the state
Scrum Poker App

Showing Training Sessions

- Load training appointments
- Show training appointments
- Show loading error message
sealed class TrainingViewState {

data class Loading : TrainingViewState()

data class Data(val trainings: List<Trainings>) : TrainingViewState()

data class Error(val error: Throwable) : TrainingViewState()
}
Unidirectional Flow
How do I get a state into my view?
Connecting the parts
User & Intent

- User
- Intent
- Model
- View

- Updates
- Manipulates
- Sees
- Interacts
interface TrainingView {

  fun loadIntent(): Observable<Boolean>
  fun pullToRefreshIntent(): Observable<Boolean>

  fun render(state: TrainingViewState)
}
interface TrainingView {

    fun loadIntent(): Observable<Boolean>
    fun pullToRefreshIntent(): Observable<Boolean>

    fun render(state: TrainingViewState)
}

31 Stefan Nägele / NovaTec Consulting GmbH
class TrainingFragment : Fragment(), TrainingView {

    private var loading = PublishSubject.create<Boolean>()

    override fun onResume() {
        super.onResume()
        loading.onNext(true)
    }

    override fun loadIntent(): Observable<Boolean> = loading

    override fun pullToRefreshIntent(): Observable<Boolean> = 
      RxSwipeRefreshLayout.refreshes(swipeRefreshLayout).map { _ -> true }
class TrainingPresenter() : MviPresenter<TrainingView>() {

  override fun attachView(view: TrainingView) {
    val loading = view.loadingIntent()
    val pullRefresh = view.pullToRefreshIntent()
    val allIntents = Observable.merge(loading, pullRefresh)

    disposable = allIntents
      .startWith(TrainingViewState.Loading())
      .onErrorReturn(TrainingViewState::Error)
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe { // We are listening for intents }
  }
}
class TrainingPresenter() : MviPresenter<TrainingView>() {

    override fun attachView(view: TrainingView) {
        val loading = view.loadingIntent()
        val pullRefresh = view.pullToRefreshIntent()
        val allIntents = Observable.merge(loading, pullRefresh)

        disposable = allIntents
            .startWith(TrainingViewState.Loading())
            .onErrorReturn(TrainingViewState::Error)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe {
                // We are listening for intents
            }
    }
}
class TrainingPresenter() : MviPresenter<TrainingView>() {

  override fun attachView(view: TrainingView) {
    val loading = view.loadingIntent()
    val pullRefresh = view.pullToRefreshIntent()
    val allIntents = Observable.merge(loading, pullRefresh)

    disposable = allIntents
      .startWith(TrainingViewState.Loading())
      .onErrorReturn(TrainingViewState::Error)
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe { // We are listening for intents }
  }
}
Intent & Model

Model

Updates
Manipulates

View

Sees

User

Interacts

Intent
Manipulating the Model

class TrainingPresenter(private val interactor: TrainingInteractor) : MviPresenter<TrainingView>() {

    override fun attachView(view: TrainingView) {
        val loading = view.loadingIntent().flatMap { interactor.getTrainings() }
        val pullRefresh = view.pullToRefreshIntent().flatMap { interactor.getTrainings() }
        val allIntents = Observable.merge(loading, pullRefresh)

        disposable = allIntents
            .startWith(TrainingViewState.Loading())
            .onErrorReturn(TrainingViewState::Error)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe { // We are listening for intents }
    }
}
class TrainingInteractor {

    fun getTrainings(): Observable<TrainingViewState> =
        Client
            .trainings()
            .toObservable()
            .map<TrainingViewState> { TrainingViewState.Data(it.trainings) }

}

object Client : RssFeedApi by Retrofit.Builder()
    .baseUrl(URL.instance)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.createWithScheduler(Schedulers.io()))
    .build()
    .create(RssFeedApi::class.java)
Model & View

- Model
- View
- Intent
- User

Updates
Manipulates
Sees
Interacts
class TrainingPresenter(private val interactor: TrainingInteractor) : MviPresenter<TrainingView>() {

    override fun attachView(view: TrainingView) {
        val loading = view.loadingIntent().flatMap { interactor.getTrainings() }
        val pullRefresh = view.pullToRefreshIntent().flatMap { interactor.getTrainings() }
        val allIntents = Observable.merge(loading, pullRefresh)

        disposable = allIntents
            .startWith(TrainingViewState.Loading())
            .onErrorReturn { TrainingViewState.Error }
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(view::render)
    }
}

interface TrainingView {
    fun render(state: TrainingViewState)
}
class TrainingFragment : Fragment(), TrainingView {

    override fun render(state: TrainingViewState) {
        when (state) {
            is Loading -> renderLoading()
            is Data -> { renderData(state.trainings) }
            is Error -> renderError()
        }
    }

    private fun renderLoading() {
        progressBar.visibility = View.VISIBLE
        trainingsList.visibility = View.GONE
        errorView.visibility = View.GONE
    }

}
Life isn't always easy, but it's simple

— Demi Moore
State Reducers

It's freaking easy!
— John Petrucci
class TrainingPresenter(private val interactor: TrainingInteractor, 
    private val reducer: TrainingReducer) : MviPresenter<TrainingView>() {

    override fun attachView(view: TrainingView) {

        val loading = view.loadingIntent().flatMap { interactor.getTrainings() }
        val pullRefresh = view.pullToRefreshIntent().flatMap { interactor.getTrainings() }
        val allIntents = Observable.merge(loading, pullRefresh)

        disposable = allIntents
            .startWith(TrainingViewState.Loading())
            .onErrorReturn(TrainingViewState::Error)
            .scan(TrainingViewState.Loading()) { state, result -> reducer.reduce(state, result) }
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(view::render)
    }
}
sealed class TrainingResult {

    object Loading: TrainingResult()

    data class LoadingError(val error: Throwable): TrainingResult()

    data class LoadingComplete(val trainings: List<Trainings>): TrainingResult()
}
class TrainingReducer {

    fun reduce(state: TrainingViewState, result: TrainingResult): TrainingViewState =
        when (result) {
            is TrainingResult.Loading -> TrainingViewState.Loading()
            is TrainingResult.LoadingError -> TrainingViewState.Error(result.error)
            is TrainingResult.LoadingComplete ->
                TrainingViewState.Data(trainings = aggregate(state.trainings, result.trainings))
        }

    private fun aggregate(previousTrainings: List<Training>, trainings: List<Training>): TrainingViewState {
        // I now pronounce you husband and wife 😛 💁‍♀️ 💁‍♂️ 💖
    }
}
Do I actually need MVI? 😐 😑
Resolving the state problem

- Traceability is increased by the unidirectional flow
- State triggers which view components are shown
- Immutable state objects

```kotlin
class TrainingFragment : Fragment(), TrainingView {

    override fun render(state: TrainingViewState) {
        when (state) {
            is Loading -> renderLoading()
            is Data -> { renderData(state.trainings) }
            is Error -> renderError()
        }
    }
}
```
null

Debuggable Workflow

class TrainingPresenter(...) : MviPresenter<TrainingView>() {

    override fun attachView(view: TrainingView) {
        val loading = view.loadingIntent()
            .doOnNext { Logger.intent("Intent: load trainings")}
            .flatMap { interactor.getTrainings() }

disposable = Observable.merge(loading, pullRefresh)
    .startWith(TrainingViewState.Loading())
    .onErrorReturn(TrainingViewState::Error)
    .scan(TrainingViewState.Loading()) { state, result -> reducer.reduce(state, result) }
    .doOnNext { Logger.state("State: ", it)}
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(view::render)
    }
}

49 Stefan Nägele / NovaTec Consulting GmbH
Debuggable Workflow

D/Intent: load trainings
D/State:
{
    "trainings": [
    {
        "description" : "14.05. - 15.05.2018, SCRUMevents,, München ",
        "link": "https://www.scrum-schulungen-stuttgart.de/professional-scrum-product-owner-paulaner-braeuhaus-muenchen",
        "title": "Professional Scrum Product Owner"
    }
    ]
}

50 Stefan Nägele / NovaTec Consulting GmbH
Testability

@RunWith(RxJavaUnitRunner::class) // Custom MockitoRunner
class TrainingPresenterTest {

    val captor = argumentCaptor<TrainingViewState>()

    val subject = PublishSubject.create<Boolean>()

    val view: TrainingFragment = mock { on { loadingIntent() } doReturn subject }

    val cut: TrainingPresenter = TrainingPresenter()
}

51  Stefan Nägele / NovaTec Consulting GmbH
@RunWith(RxJavaUnitRunner::class) class TrainingPresenterTest {

    @Test fun renderData() {
        Server.dispatcher = { MockResponse().setResponseCode(HTTP_OK).setBody(...) }
        cut.attachView(view)
        subject.onNext(true)

        verify<TrainingFragment>(view, times(2)).render(captor.capture())
        expect(captor.firstValue).toBeInstanceOf<TrainingViewState.Loading>()
        expect((captor.secondValue as Data).trainings).toContain(Training("title1"...))
    }
}
@RunWith(RxJavaUnitRunner::class) class TrainingPresenterTest {

    @Test fun renderError() {
        Server.dispatcher = { MockResponse().setResponseCode(HTTP_INTERNAL_ERROR) }
        cut.attachView(view)
        subject.onNext(true)

        verify<TrainingFragment>(view, times(2)).render(captor.capture())
        expect(captor.firstValue).toBeInstanceOf<TrainingViewState.Loading>()
        expect(captor.secondValue).toBeInstanceOf<TrainingViewState.Error>()
    }
}
Testability

Model

View

Intent

User

Updates

Manipulates

Sees

Interacts

Stefan Nägele / NovaTec Consulting GmbH
Orientation change / Interruption

Persisting state with ease

class TrainingFragment : Fragment(), TrainingView {

    override fun render(state: TrainingViewState) {
        this.state = state
    }

    override fun onSaveInstanceState(out: Bundle) {
        super.onSaveInstanceState(outState)
        out.putParcelable("MyStateKey", state)
    }

    override fun onCreate(saved: Bundle) {
        super.onCreate(saved)

        val initState = state?.getParcelable("MyStateKey")
        presenter = TrainingPresenter(TrainingInteractor(), TrainingReducer(initState))
    }
}
Independent UI Components

- Whenever an Event X happens, presenter sends information to the business logic
- Presenters observing the same business logic for the same state
Drawbacks
Drawbacks 😞

- Requires a lot of boilerplate code
  - Intents, States, Results
- Reducers are getting big
  - Casting in Java is a pain in the ass
  - Readability with Switch Cases in Java as well
private val reducer = BiFunction { previousState: TasksViewState, result: TasksResult ->
    when (result) {
        is LoadTasksResult -> when (result) {
            is LoadTasksResult.Success -> {
                val filterType = result.filterType ?: previousState.tasksFilterType
                val tasks = filteredTasks(result.tasks, filterType)
                previousState.copy(isLoading = false, tasks = tasks, tasksFilterType = filterType)
            }
            is LoadTasksResult.Failure -> previousState.copy(isLoading = false, error = result.error)
            is LoadTasksResult.InFlight -> previousState.copy(isLoading = true)
        }
        is CompleteTaskResult -> when (result) {
            is CompleteTaskResult.Success -> {
                previousState.copy(taskComplete = true, tasks = filteredTasks(result.tasks, previousState.tasksFilterType)
            }
            is CompleteTaskResult.Failure -> previousState.copy(error = result.error)
            is CompleteTaskResult.InFlight -> previousState.copy(taskComplete = false)
            is CompleteTaskResult.HideUiNotification -> previousState.copy(taskComplete = false)
        }
    }
}
Drawbacks 😞

- Requires a lot of boilerplate code
  - Intents, States, Results
- Reducers are getting big
  - Casting in Java is a pain in the ass
  - Readability with Switch Cases in Java as well
- Requires RxJava
Benefits

- Immutability and unidirectional data flow
- Solving the state problem
  - State easy to control, trace and debug
  - Persisting the state (orientation change, process death)
  - Controlling different UI components
- Increased testability
References

Hannes Dorfmann - Reactive Apps with Model-View-Intent
Jake Wharton - Managing State with RxJava
Benoît Quenaudon @droidconNYC - Model-View-Intent for Android
Garima Jain @droidconbos - The Curious Case Of Yet Another Pattern
André Staltz - Unidirectional User Interface Architectures
Stefan Nägele

I am Stefan

- Consultant @novatecgmbh
- Twitter: @Ethanoljesus
- Blog: blog.novatec-gmbh.de
- Zxing: Stefan_Naegele7

Thanks for your attention!