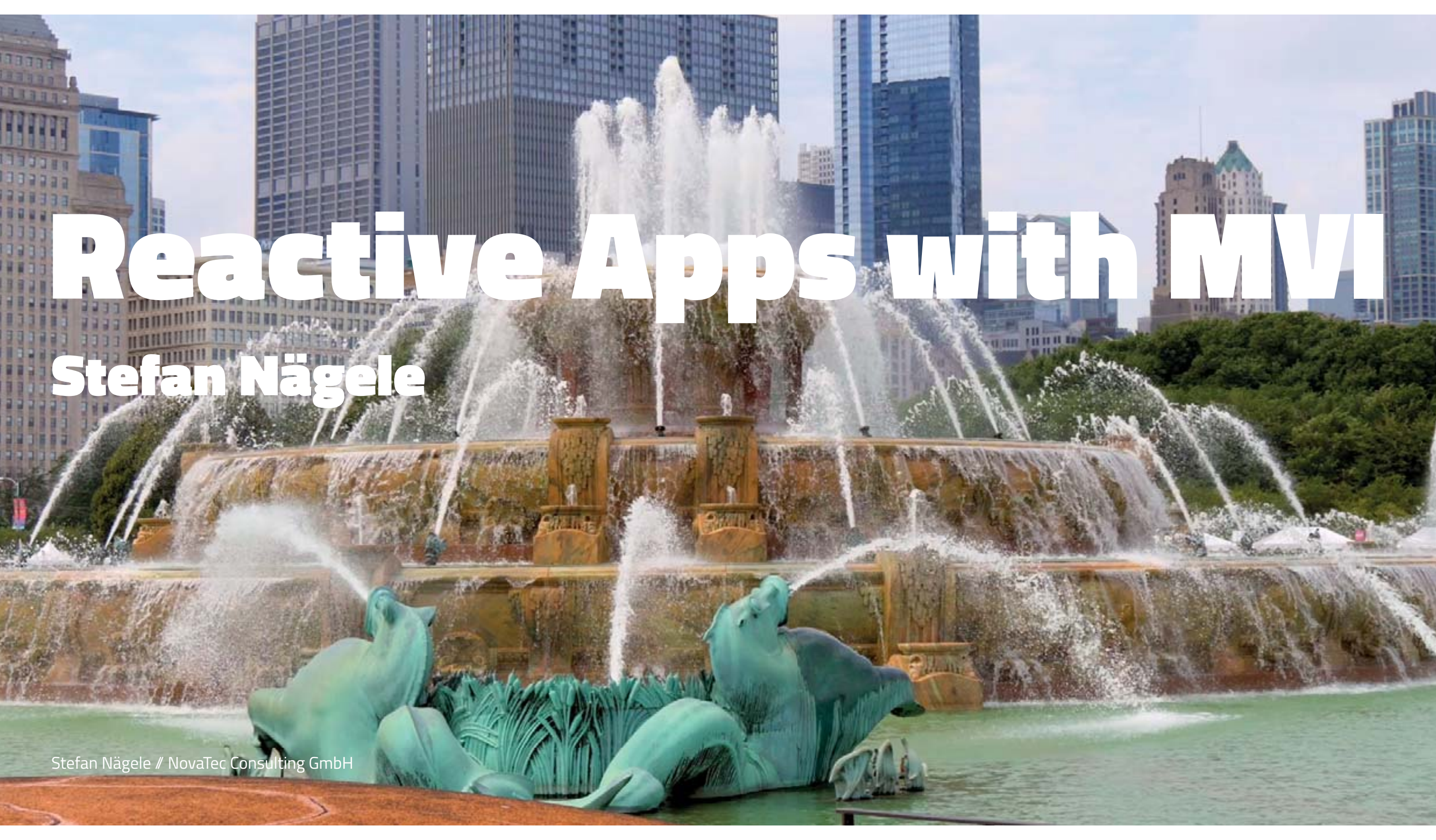

Reactive Apps with MVI
Stefan Nägele
Stefan Nägele // NovaTec Consulting GmbH

Stefan Nägele

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

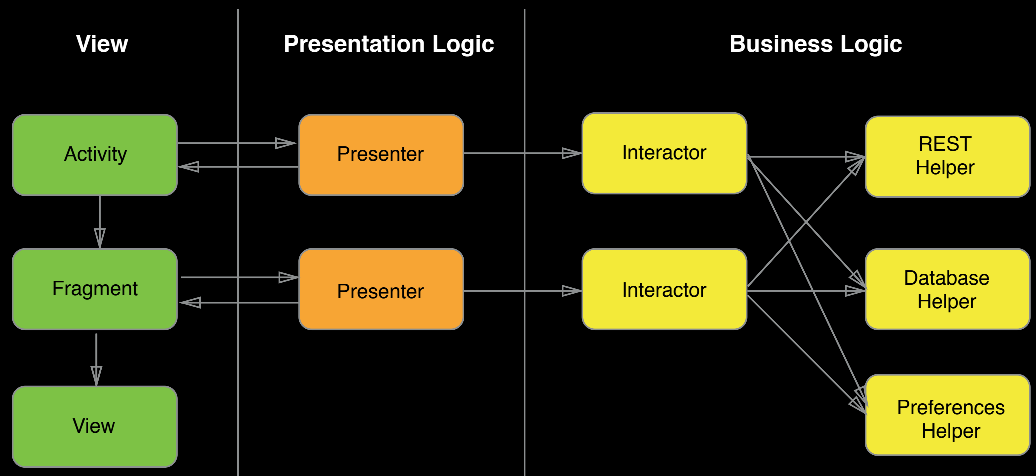
I am here to tell you how to make reactive (Android) apps with MVI

Way to reactiveness
The development of Android architectures
RxJava
How to master reactiveness with MVI

The early days

Android Architectures

Nowadays



Android Architectures

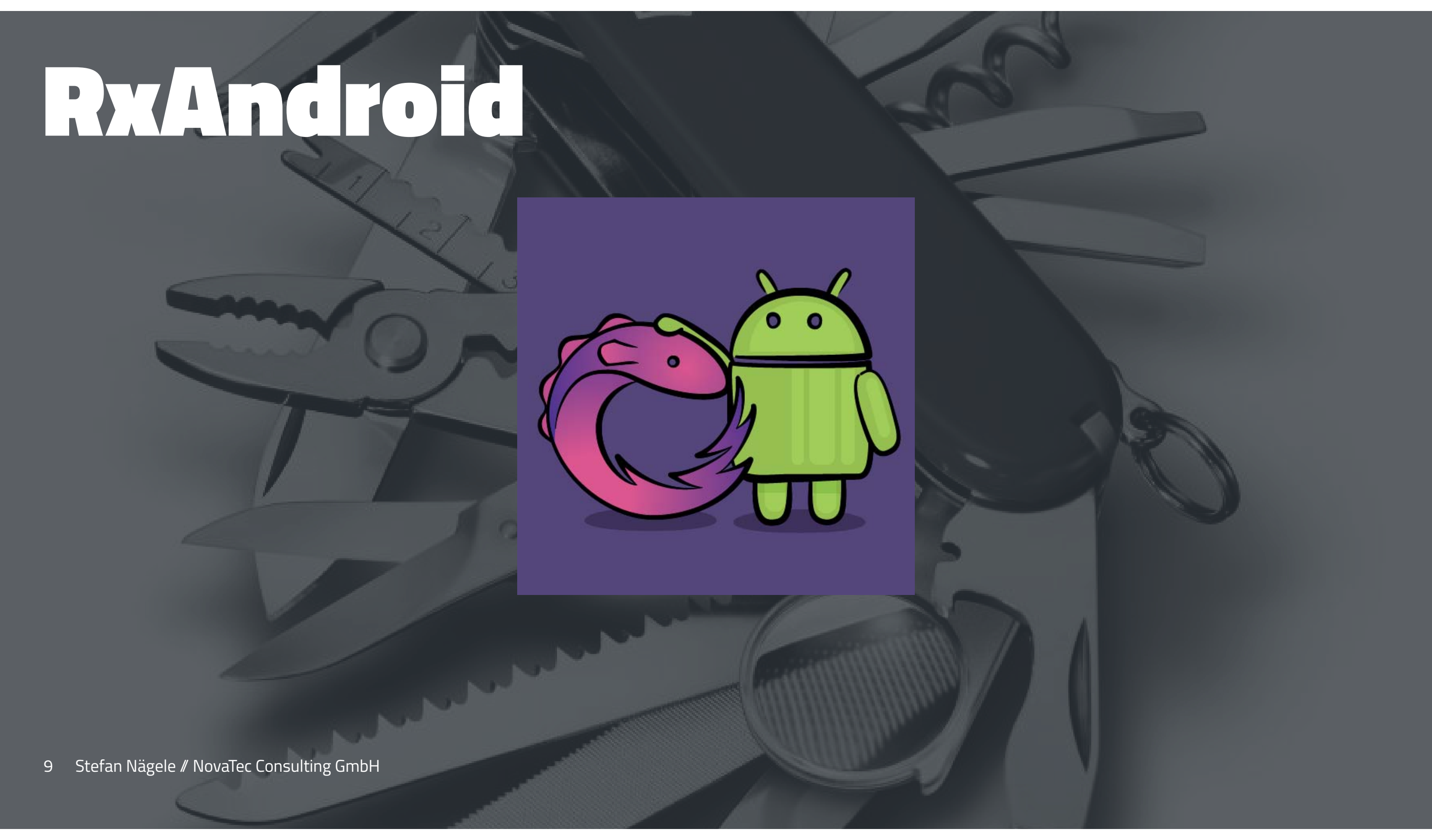
Nowadays

- Separation of concerns
  - Expandable
  - Maintanable
  - 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 */);
```

UI

- 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.
- Rx Preferences - Reactive SharedPreferences for Android
- RxCache - Reactive caching library for Android and Java
- RxCupboard - 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.
- SQLBrite - 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

The Data Flow Pitfall

```
RxView
    .clicks(submitView)
    .doOnNext {
        submitView.setEnabled(false);
        progressView.setVisibility(VISIBLE);
    }
    .flatMap { interactor.setName(nameView.text.toString()) }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(/* an Observer */)
```

The Data Flow Pitfall

```
RxView
    .clicks(submitView)
    .doOnNext {
        submitView.setEnabled(false);
        progressView.setVisibility(VISIBLE);
    }
    .flatMap { interactor.setName(nameView.text.toString()) }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(/* an Observer */)
```

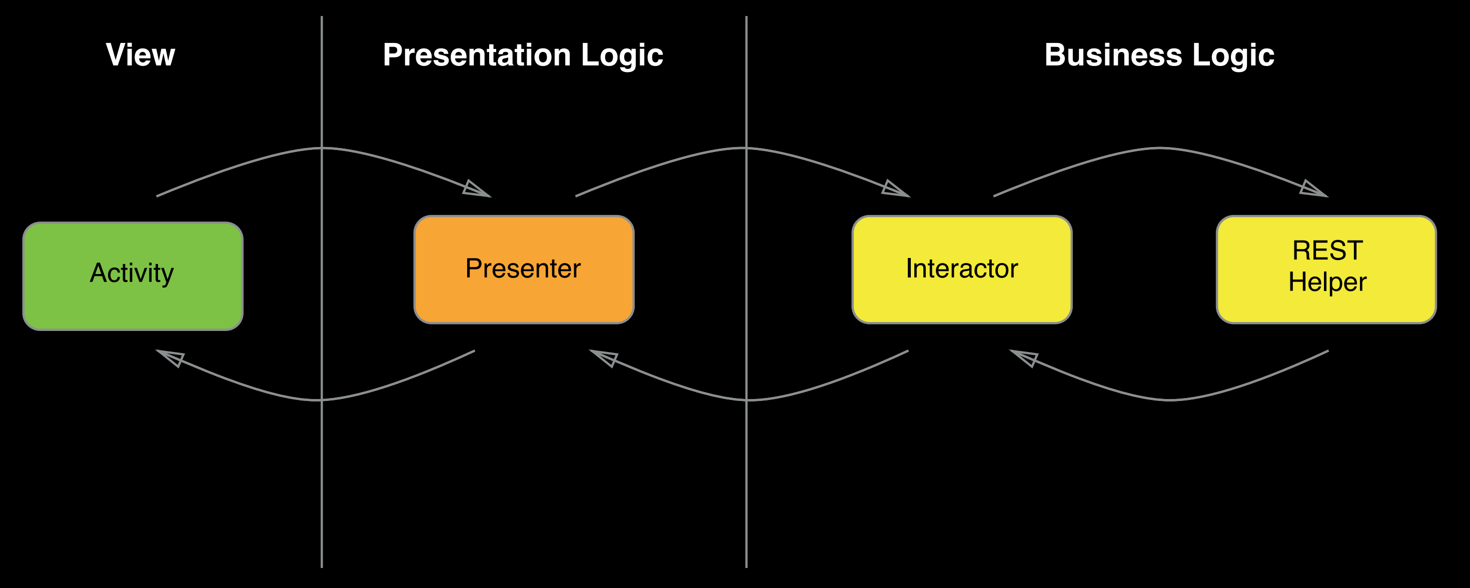
The Data Flow Pitfall

```
RxView
    .clicks(submitView)
    .doOnNext {
        submitView.setEnabled(false);
        progressView.setVisibility(VISIBLE);
    }
    .flatMap { interactor.setName(nameView.text.toString()) }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(/* an Observer */)
```

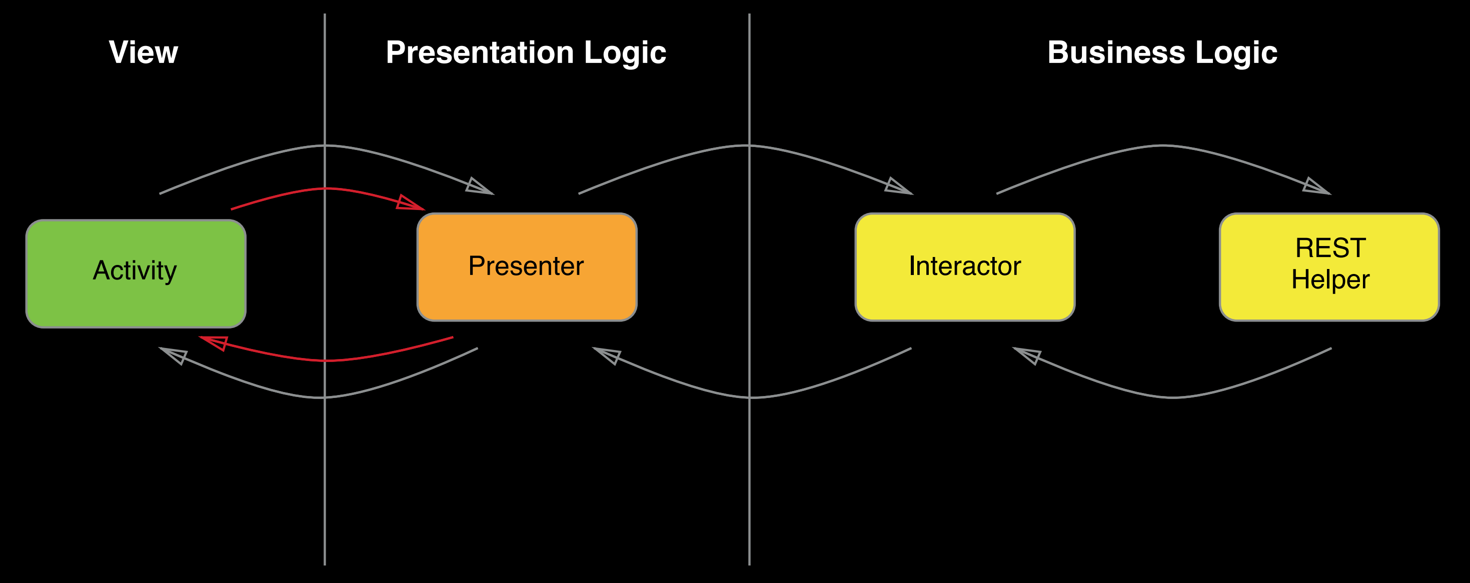
The Data Flow Pitfall

```
RxView
    .clicks(submitView)
    .doOnNext {
        submitView.setEnabled(false);
        progressView.setVisibility(VISIBLE);
    }
    .flatMap { interactor.setName(nameView.text.toString()) }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(/* an Observer */)
```

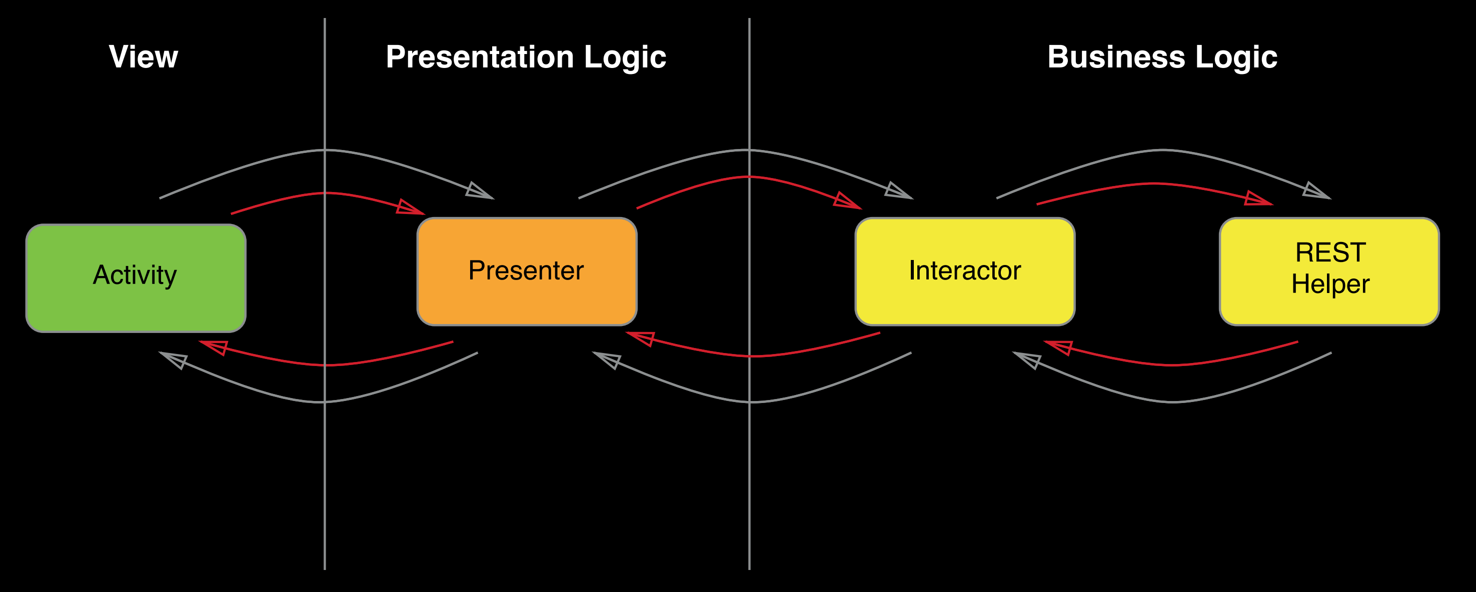
The Data Flow Pitfall



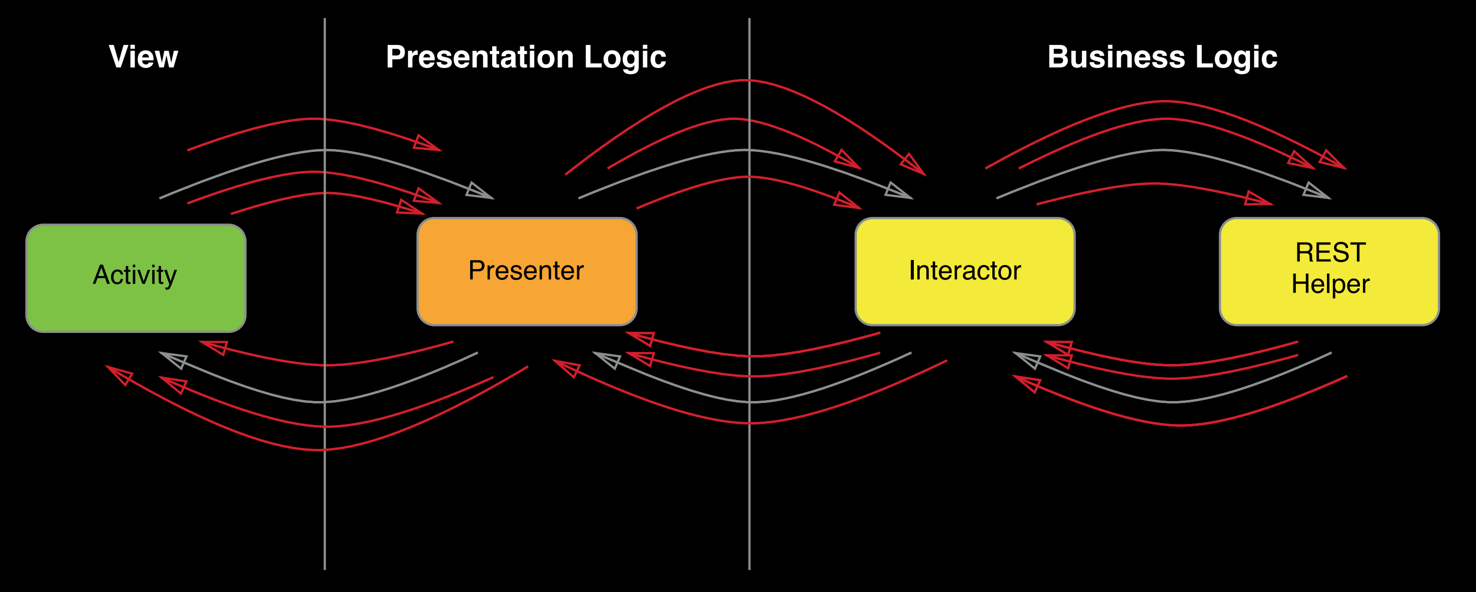
The Data Flow Pitfall



The Data Flow Pitfall



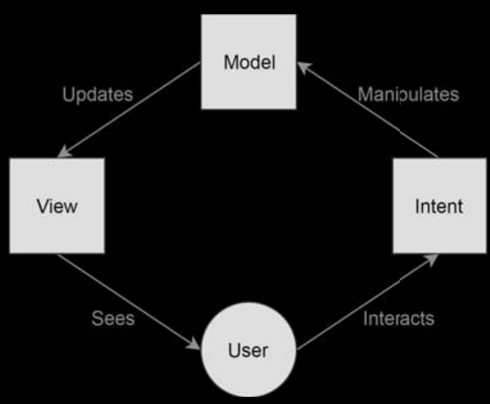
The Data Flow Pitfall



The State Problem

Model View Intent

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



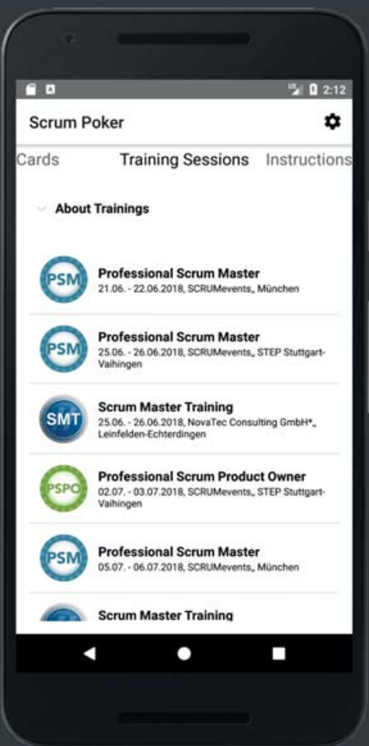
Model

- Single source of truth for...
  - ...the business Logic
  - ...and view
- Model reflects a view's state

Scrum Poker App

Showing Training Sessions

- Load training appointments
- Show training appointments
- Show loading error message



Defining a State

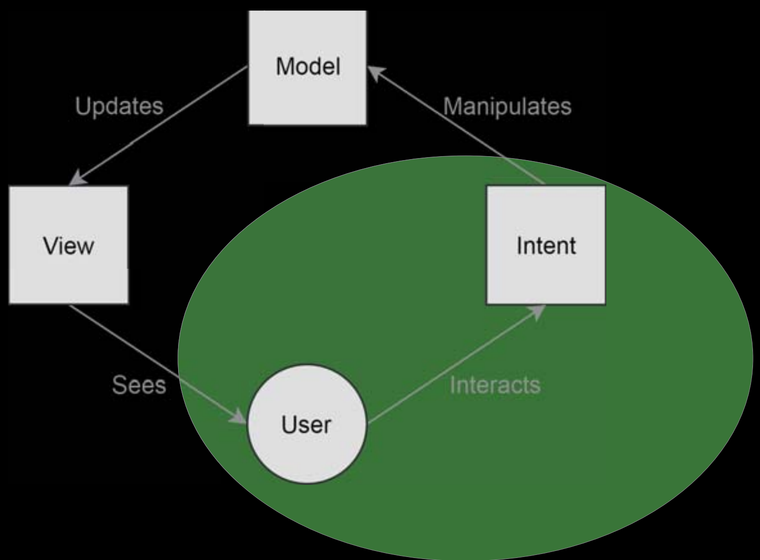
```
sealed class TrainingViewState {

    data class Loading() : TrainingViewState()

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

    data class Error(val error: Throwable) : TrainingViewState()
}
```

Unidirectional Flow

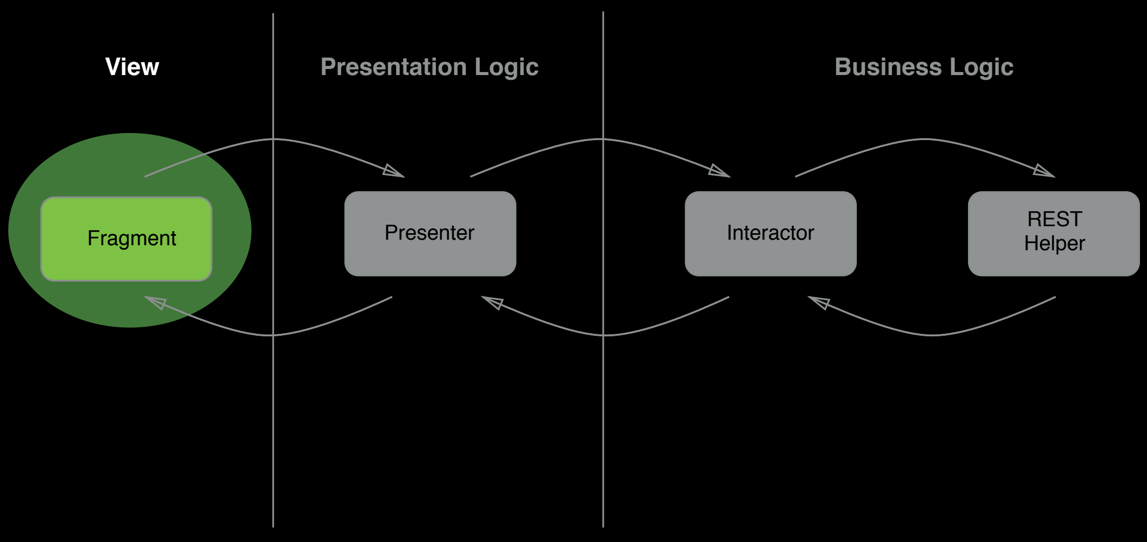
How do I get a state into my view? 🤷‍♀️

Connecting the pieces

User & Intent



User & Intent - View



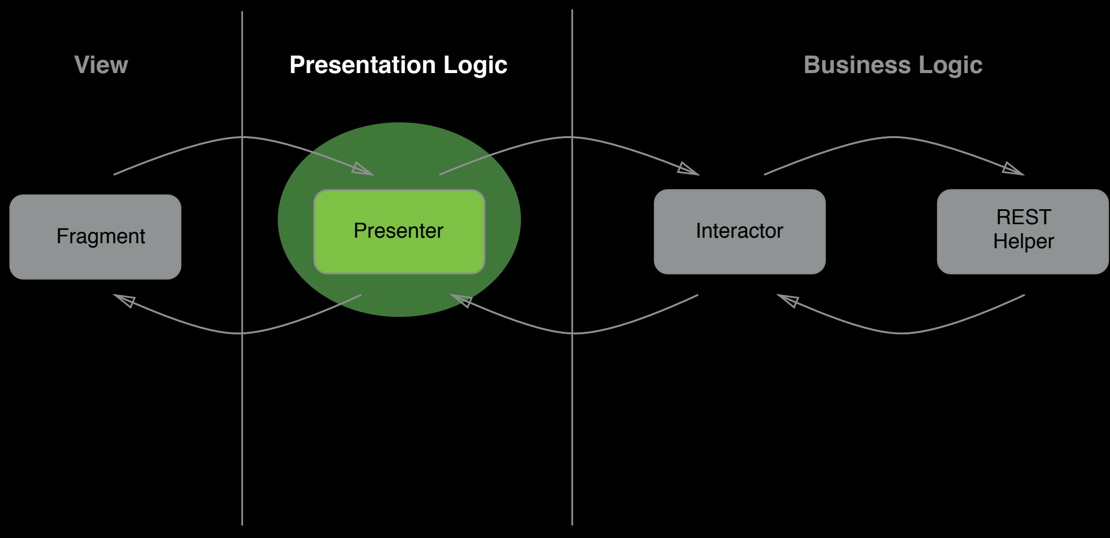
User & Intent - Intents

```
interface TrainingView {

    fun loadIntent(): Observable<Unit>
    fun pullToRefreshIntent(): Observable<Unit>
}
```

User & Intent - View

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

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

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

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

    override fun pullToRefreshIntent(): Observable<Unit> =
        RxSwipeRefreshLayout.refreshes(swipeRefreshLayout).map { Unit }
}
```

User & Intent - Presentation



Listening to 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 }
    }
}
```

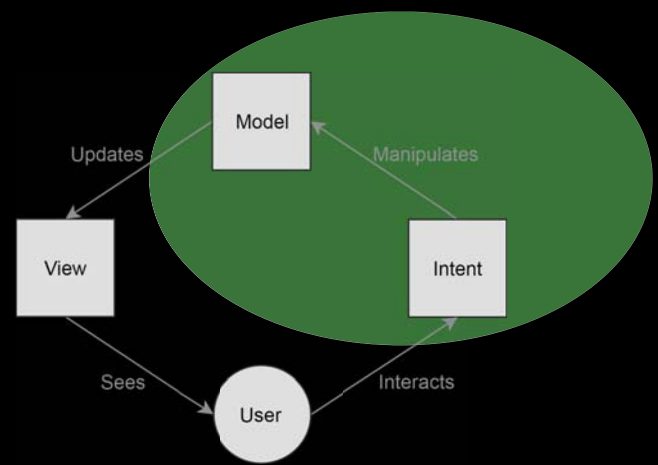
Initial State

```
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 }
    }
}
```

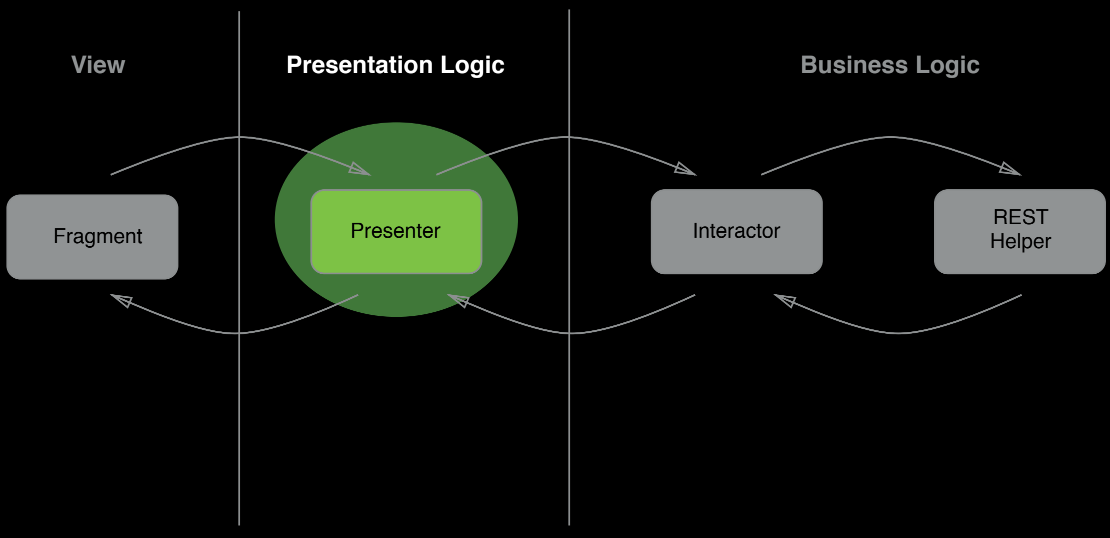
Expect the unexpected

```
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



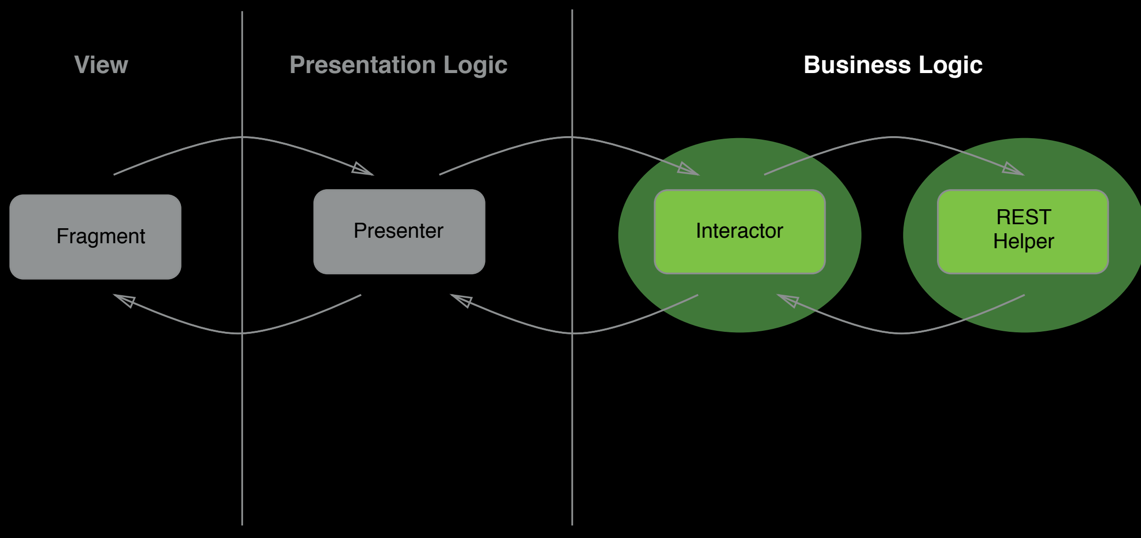
Intent & Model - Presentation



Manipulating the Model

```kotlin
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 }
    }
}
```

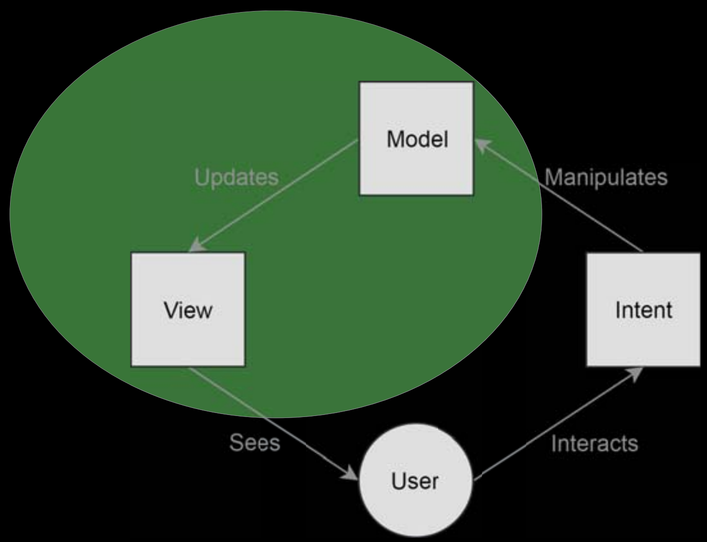
Intent & Model - Interactor



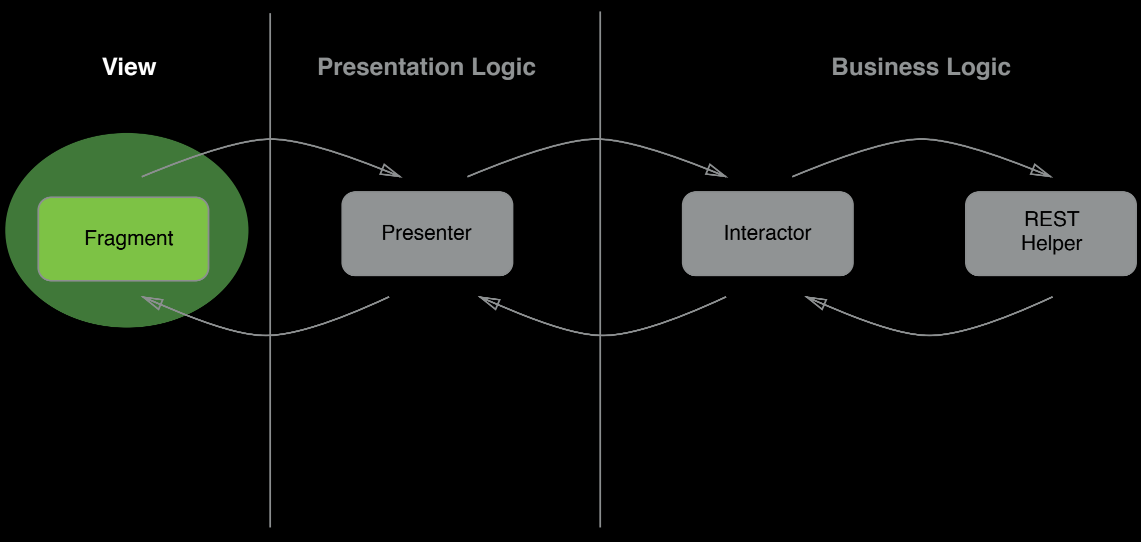
Business Logic

```
class TrainingInteractor {

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

object Client : RssFeedApi by Retrofit.Builder()
        .addCallAdapterFactory(RxJava2CallAdapterFactory.createWithScheduler(Schedulers.io()))
        .create(RssFeedApi::class.java)
```

Model & View



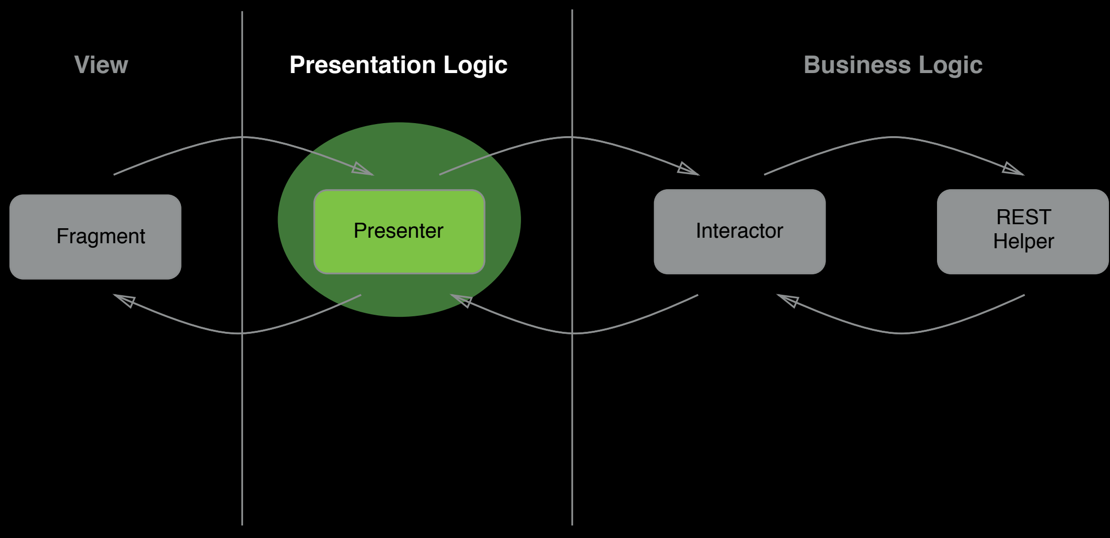
Updating the View - View



Updating the View - View

```
interface TrainingView {

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

    fun render(state: TrainingViewState)
}
```

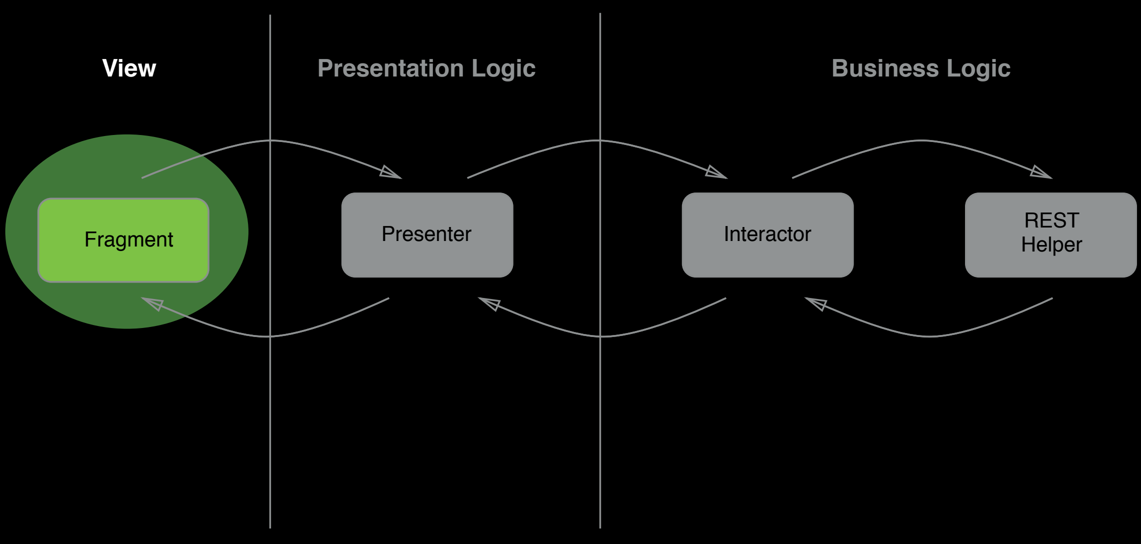
Updating the View - Presenter



Updating the View - Presenter

```
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)
    }
}
```

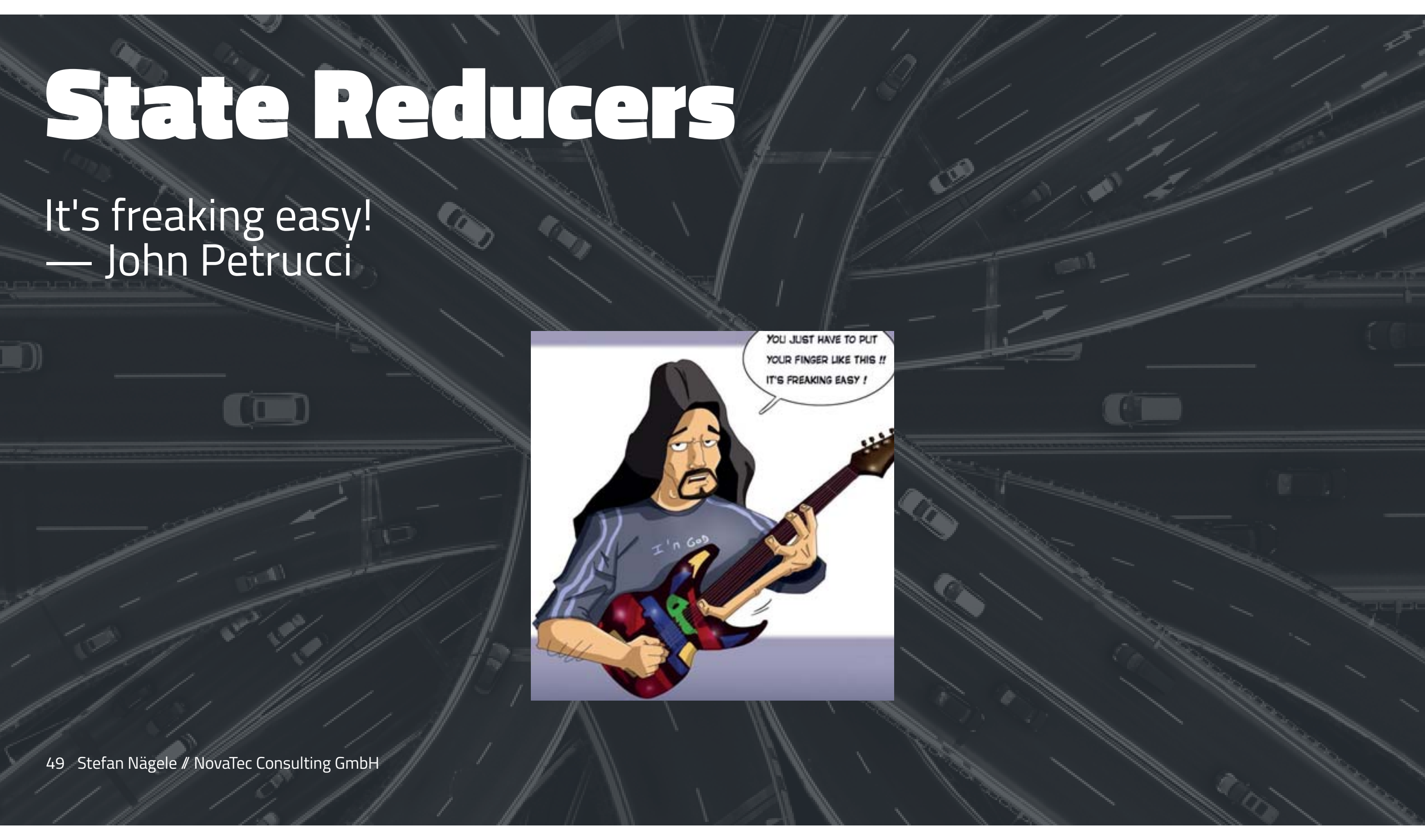
Updating the View - View



Updating the View - View

```
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
YOU JUST HAVE TO PUT
YOUR FINGER LIKE THIS !!
IT'S FREAKING EASY !
49 Stefan Nägele // NovaTec Consulting GmbH

State Reducers - Presenter

```
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)
    }
}
```

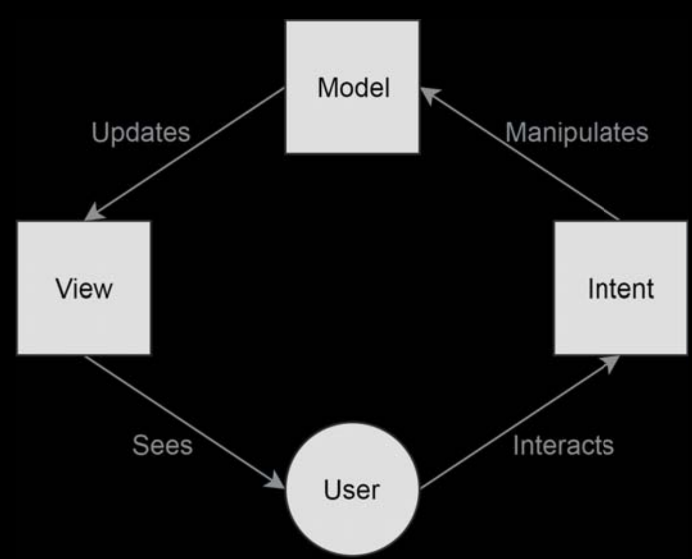
Partial State - TrainingResult

```
sealed class TrainingResult {

    object Loading: TrainingResult()

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

    data class LoadingComplete(val trainings: List<Training>): TrainingResult()
}
```

State Reducer

```
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

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

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

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)
    }
}
```

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"
        }
    ]
}
```

Testability 👌

```kotlin
@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()
}
```

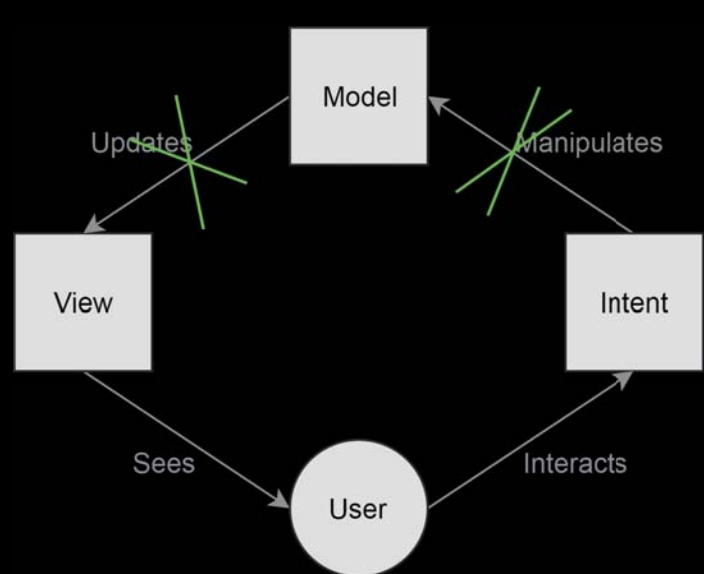
Testability 👌

```
@RunWith(RxJavaUnitRunner::class) class TrainingPresenterTest {

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

        verify<TrainingFragment>(view, times(2)).render(captor.capture())
        expect(captor.firstValue).toBeInstanceOf<TrainingViewState.Loading>()
        expect((captor.secondValue as Data).trainings).toContain(Training("title1"...))
    }
}
```

Testability 👌

```
@RunWith(RxJavaUnitRunner::class) class TrainingPresenterTest {

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

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

Testability 👌



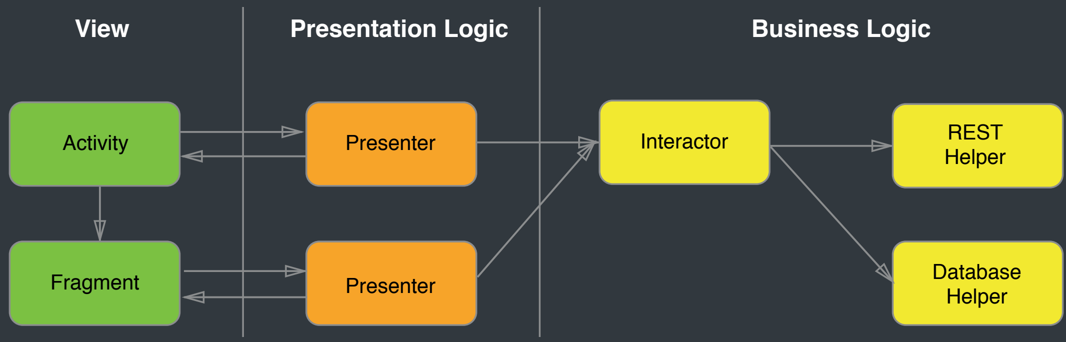
Orientation change / Interruption

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, presentation logic 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

Drawbacks - Reducers

```
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
          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

SAYAROSENROT

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

Questions?

Bibliography

- Buckingham Fountain videvo.net
- Stairway to heaven - thiswallpaper.com
- Swiss Knife - schweizer-taschenmesser.de
- Error and loading - youtube.com
- RxAndroid - koenig-media.raywenderlich.com
- Atom - atom.io
- Complex problems - tintri.com
- Darth Vader - playnation.de
- Dream Theater - A dramatic turns of events - rosenrotsaya.blogspot.com
- Melk Abbey Library en.wikipedia.org
- Questions - structuretech1.com