

MVI

New state

New object

New world

MVVM

MVI

MVC

MVP

„Simple, revolutionary, coming from the world
where $[]+[] = ""$ ”

–Someone Somewhere

Literally

now

it

is

Everything is a stream

Everything

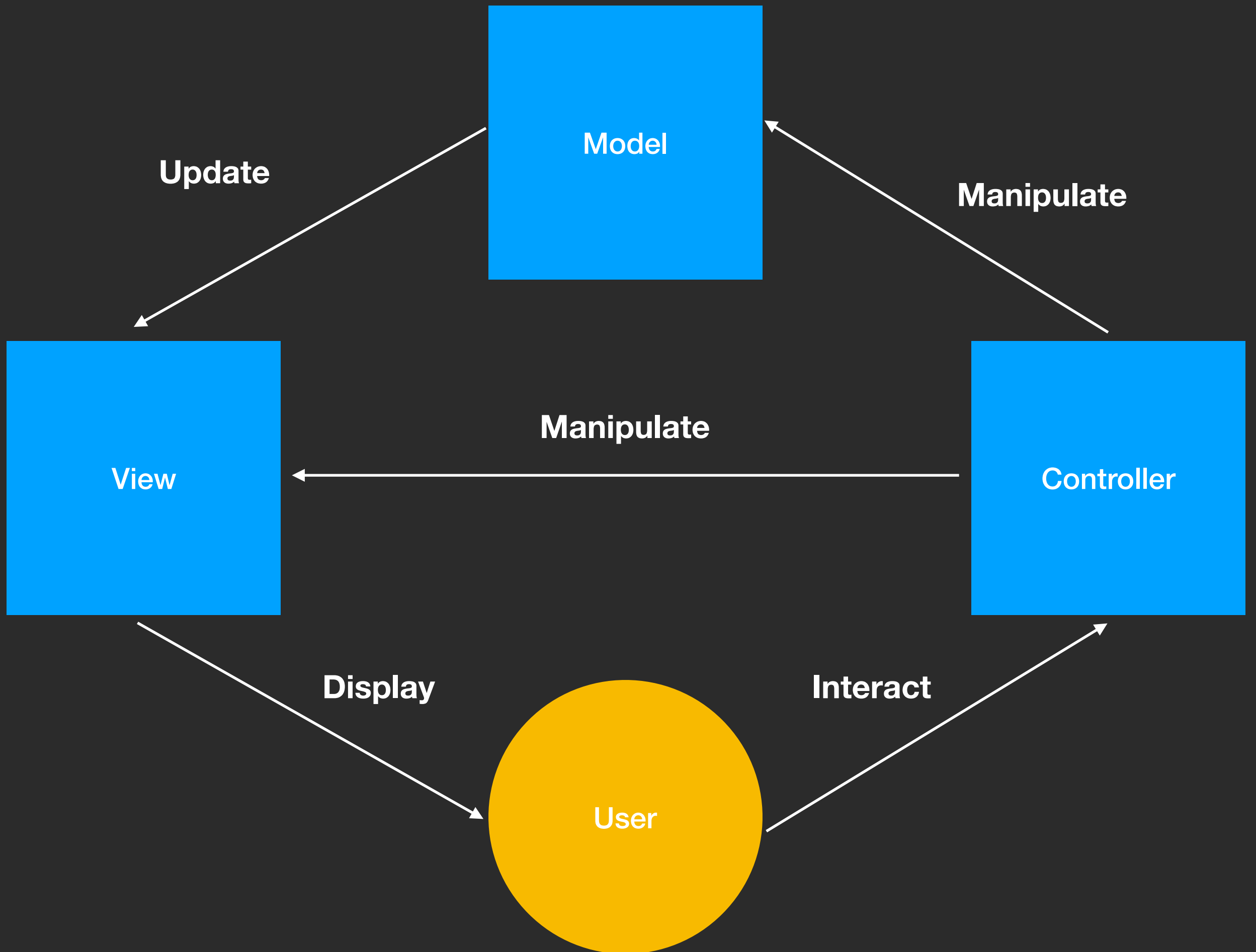


„What?! There’s such operator in RxJava?”

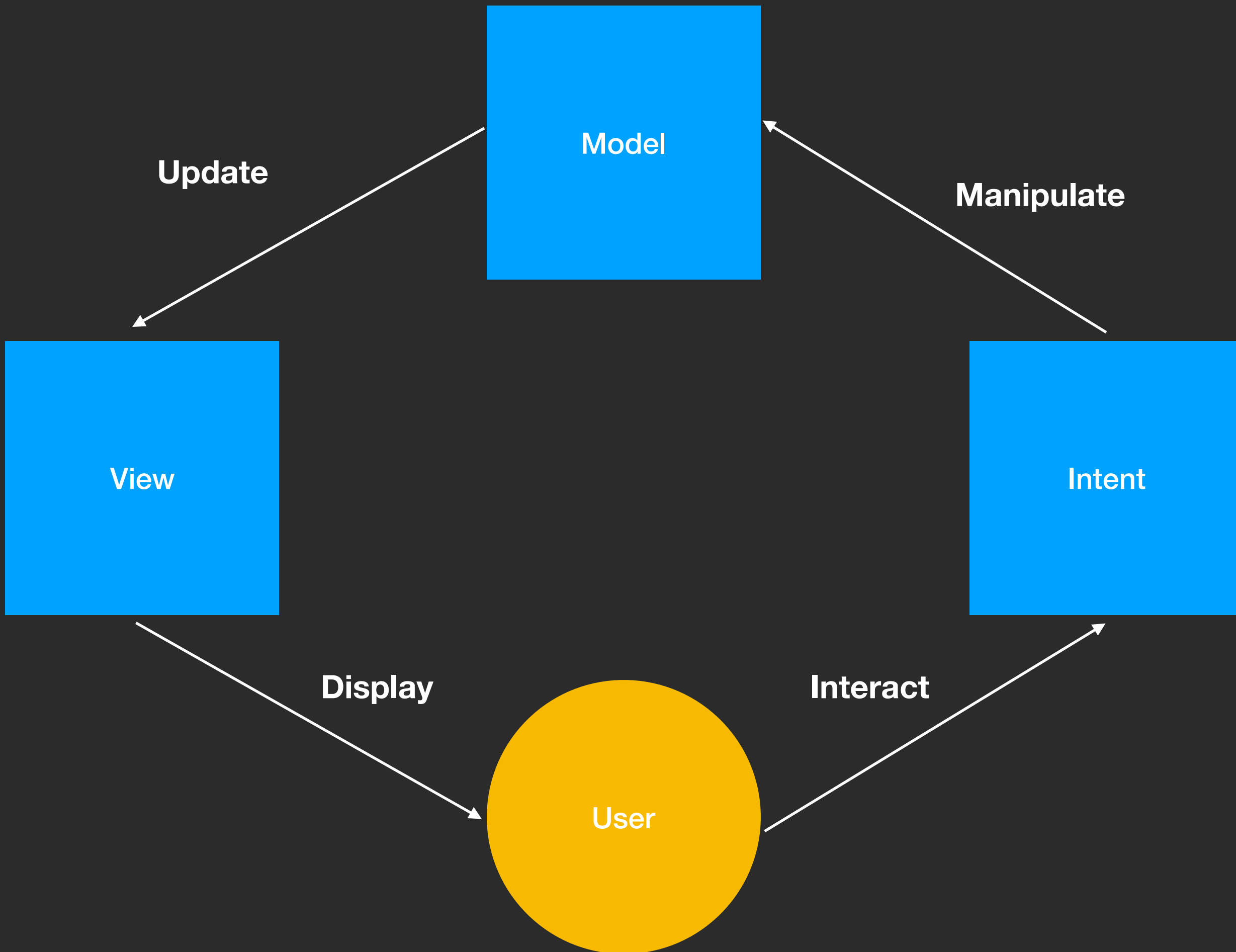
„What?! You can do that with Rx too?”

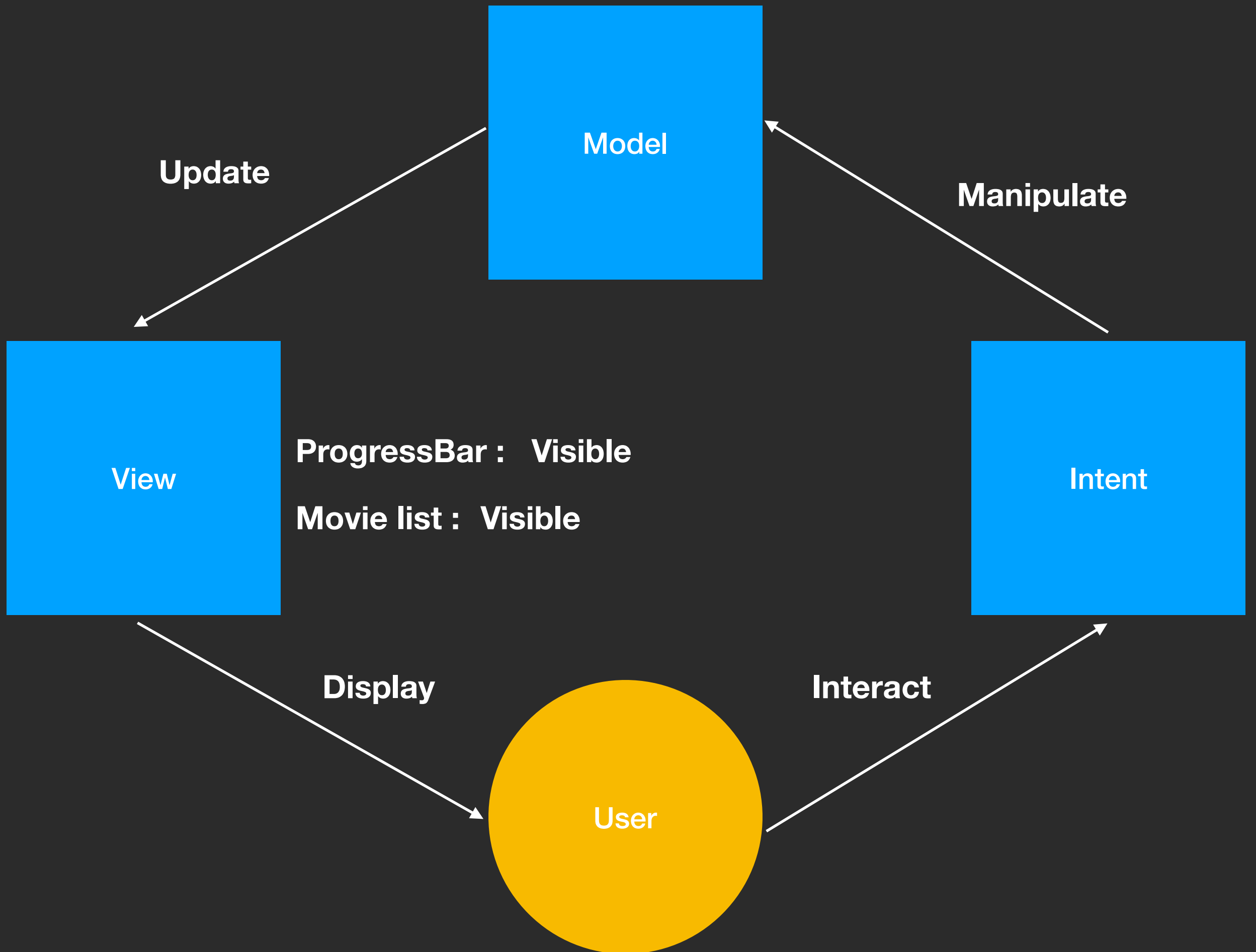
„What?! Sealed classes can do that?”

–Me



Where is the state?





MVI Sample

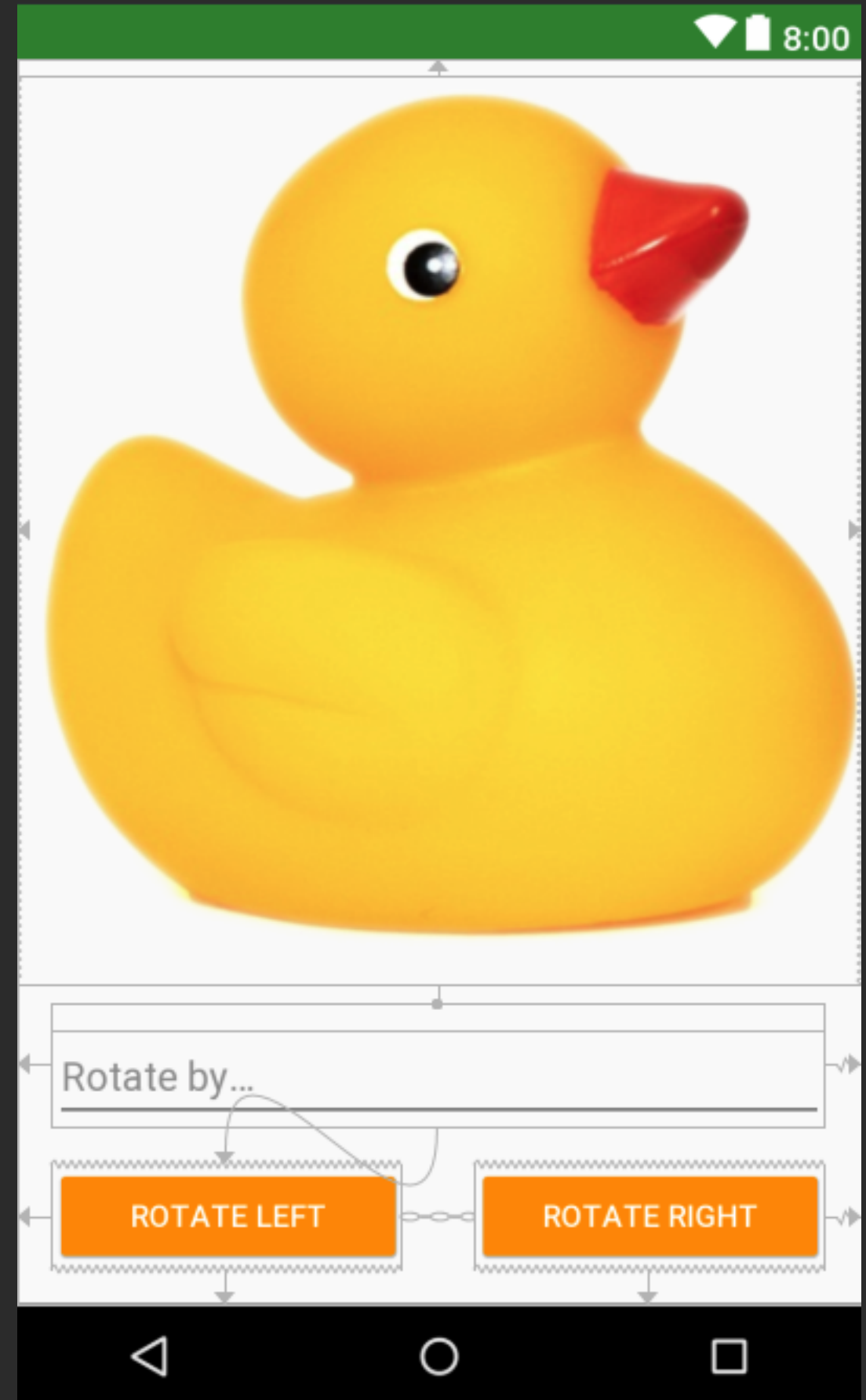
999



Show me the code!

Let's spin the duck!

- User can enter degrees
- Rotate left / Rotate right should rotate the static image
- State should be retained



```
class DuckFragment : BaseMviFragment<DuckView, DuckMviViewModel>(),
    DuckView {

    companion object {
        fun newInstance() = DuckFragment()
    }

    override val rotateLeftClicks by lazy { rotateLeft.clicks() }
    override val rotateRightClicks by lazy { rotateRight.clicks() }
    override val rotateBy by lazy { rotateByTextInput.textChanges() }

    override fun getMviViewModel() = ViewModelProviders.of(this)
        .get(DuckMviViewModel::class.java)

    override fun onCreateView(...)

    override fun render(duckViewState: DuckViewState) {
        duckImage.rotation = duckViewState.rotation
    }
}
```



```
class DuckMviViewModel : BaseMviViewModel<DuckView, DuckViewState>() {  
    override fun bindIntents() {  
        val rotateByObservable = intent { it.rotateBy }  
            .map { it.toString().toFloatOrNull() ?: 0.0f }  
  
        val rotateLeftObservable = intent { it.rotateLeftClicks }  
            .withLatestFrom(rotateByObservable) { _, rotation -> rotation }  
            .map { -it }  
  
        val rotateRightObservable = intent { it.rotateRightClicks }  
            .withLatestFrom(rotateByObservable) { _, rotation -> rotation }  
  
        val stateObservable = Observable.merge(  
            rotateLeftObservable,  
            rotateRightObservable  
        )  
            .scan(DuckViewState()) { oldState, rotationChange ->  
                oldState.copy(rotation = oldState.rotation + rotationChange)  
            }  
  
        subscribeViewState(stateObservable) { view, viewState ->  
            view.render(viewState)  
        }  
    }  
}
```

MVI Sample

DUCK ACTIVITY

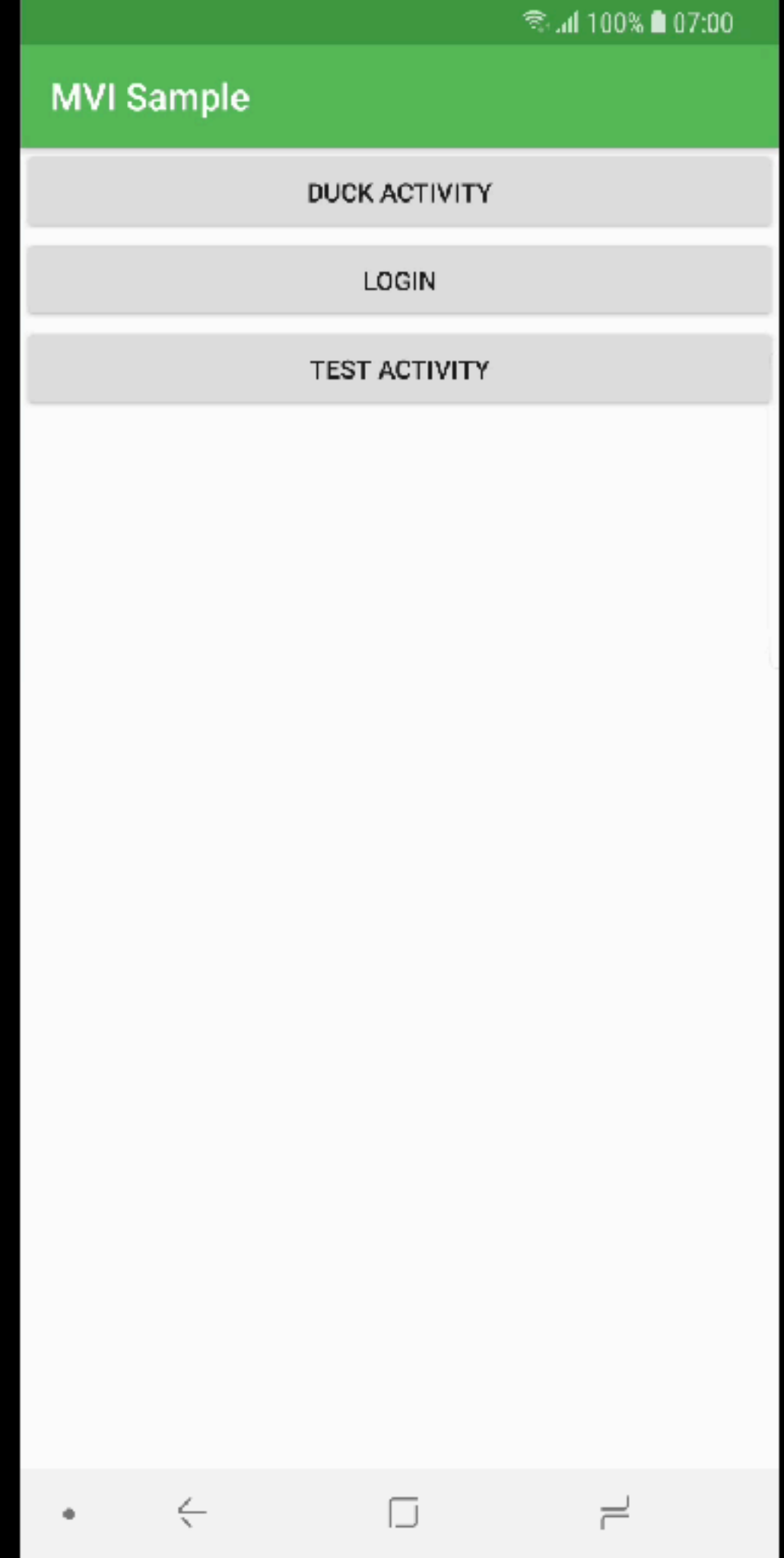
LOGIN

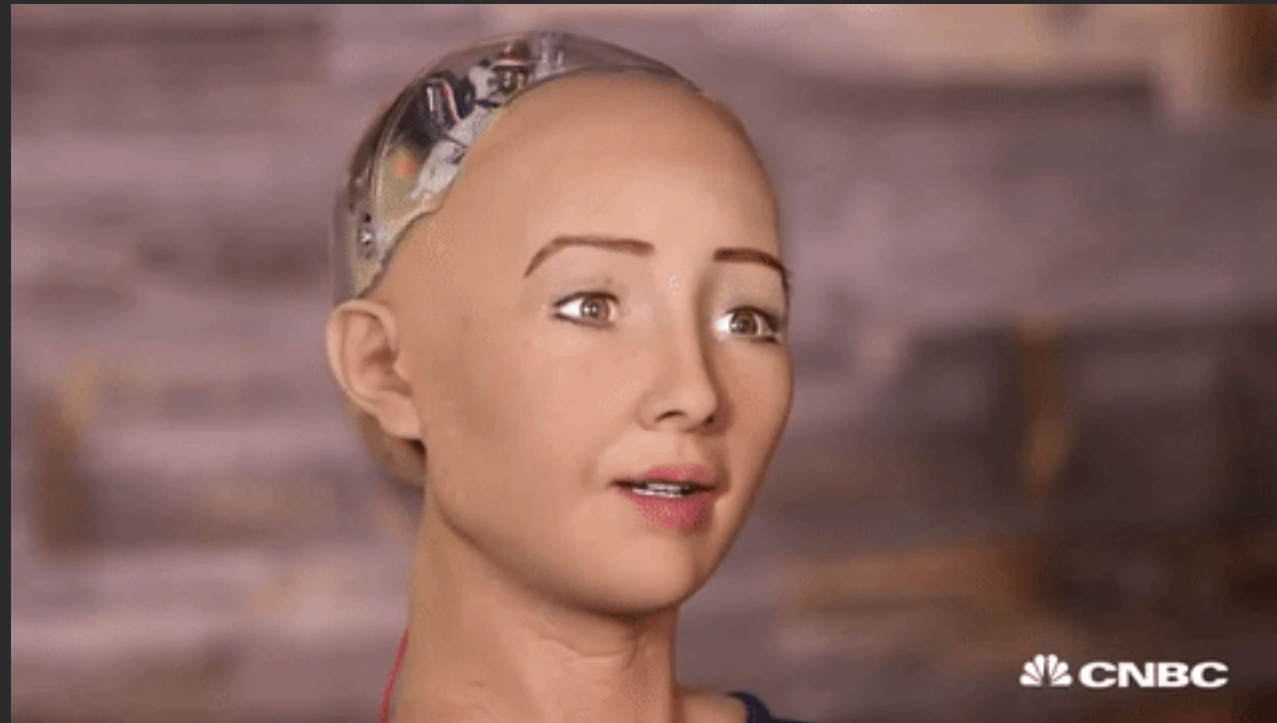
TEST ACTIVITY



Simple login screen

- Should show error when pressing login with empty fields and clear up after reentry.
- Should validate email and password before making request.
- Should show progress bar and login when correct password and mail is entered.
- Should show incorrect credentials when authentication error and clean up if any of inputs are modified.
- Should show unknown error for 2 seconds if unknown error occurs while logging in.





ViewRobot concept

```
class LoginViewRobot(private val presenter: LoginMviViewModel) :  
    MviViewRobotBase<LoginViewState>(), LoginView {  
  
    override val loginClicks: Subject<Unit> = PublishSubject.create()  
    override val emailTextChange: Subject<String> = PublishSubject.create()  
    override val passwordTextChange: Subject<String> = PublishSubject.create()  
  
    init {  
        presenter.attachView(this)  
    }  
  
    override fun render(viewState: LoginViewState) {  
        renderEvents.add(viewState)  
    }  
  
    override fun destroyView() = presenter.detachView()  
  
    fun clickLogin() = loginClicks.onNext(Unit)  
  
    fun enterEmail(email: String) = emailTextChange.onNext(email)  
  
    fun enterPassword(password: String) = passwordTextChange.onNext(password)  
}
```

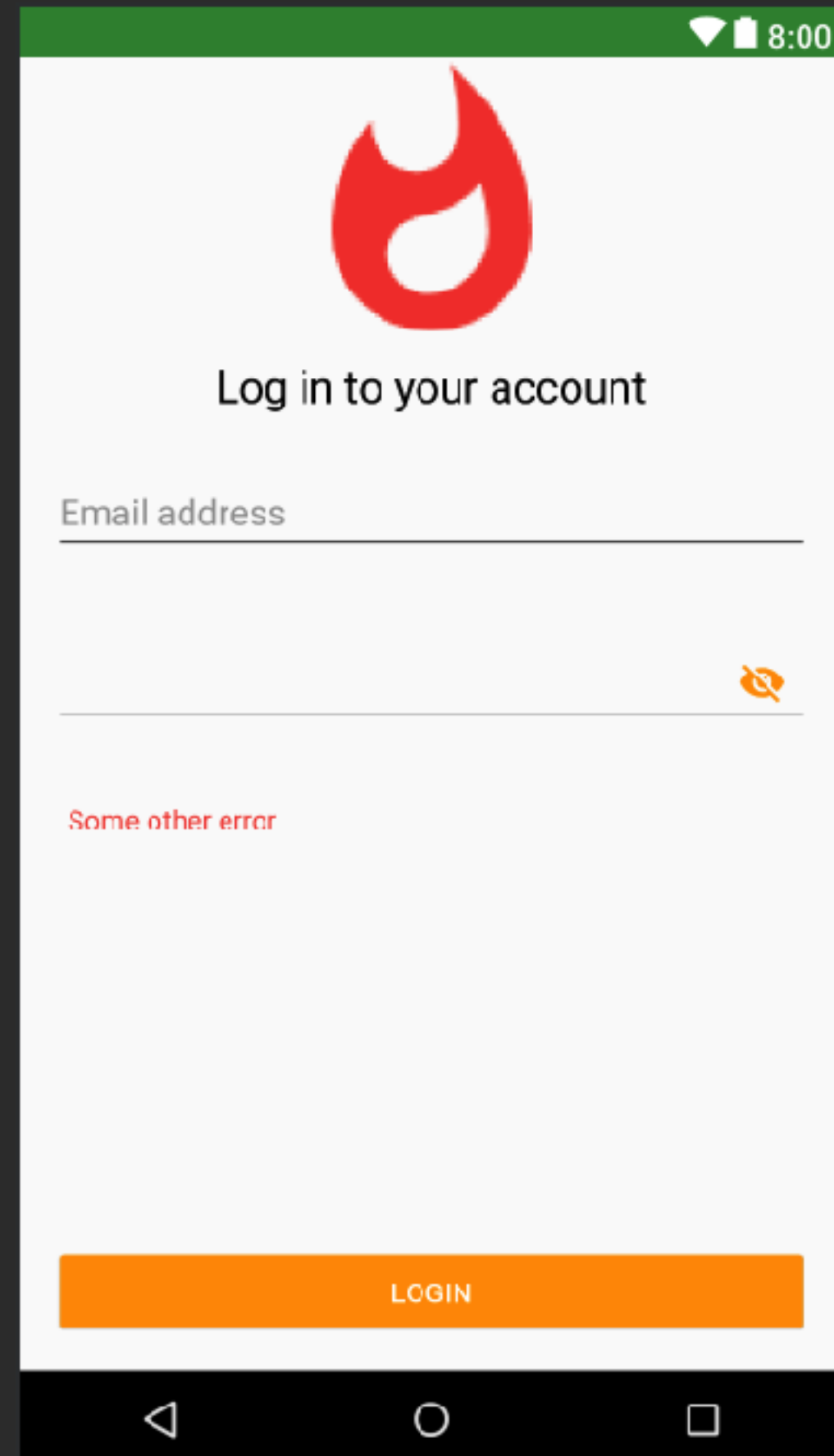
```
abstract class MviViewRobotBase<VS> {  
    protected val renderEvents = mutableListOf<VS>()  
  
    fun assertViewStatesRendered(vararg viewStates: VS) {  
        Assert.assertEquals(  
            "View states list should be the same",  
            viewStates.asList(),  
            renderEvents  
        )  
    }  
  
    abstract fun destroyView()  
}
```

```
sealed class LoginViewState{

data class State(
    val mailInputError: MailInputError? = null,
    val passwordInputError: PasswordInputError? =
        null,
    val credentialsError: CredentialsError? =
        null,
    val otherErrors: OtherErrors? = null,
    val progressState: ProgressState =
        ProgressState.WAITS
) : LoginViewState()

object Success : LoginViewState()

}
```



```
@Test
fun `Should show error when pressing login with empty fields and clear up
after modifying given input`() {

    loginViewRobot.enterEmail("")
    loginViewRobot.enterPassword("")
    loginViewRobot.clickLogin()
    loginViewRobot.enterEmail("a")
    loginViewRobot.enterPassword("v")

    loginViewRobot.assertViewStatesRendered(
        LoginViewState.State(),
        LoginViewState.State(
            mailInputError = MailInputError.INCORRECT,
            passwordInputError = PasswordInputError.TO_SHORT
        ),
        LoginViewState.State(
            passwordInputError = PasswordInputError.TO_SHORT
        ),
        LoginViewState.State()
    )
}
```

```
@Test
fun `Should show progressbar and login when correct password`() {
    loginViewRobot.enterEmail(CORRECT_MAIL)
    loginViewRobot.enterPassword(CORRECT_PASSWORD)
    loginViewRobot.clickLogin()
    loginSingleSubject.onSuccess(USER_TOKEN)

    loginViewRobot.assertViewStatesRendered(
        LoginViewState.State(),
        LoginViewState.State(
            progressState = ProgressState.LOGGING_IN
        ),
        LoginViewState.Success
    )

    verify(tokenRepositoryMock).saveToken(USER_TOKEN)
}
```



```
@Test
fun `Should show unknown error for 2 seconds if unknown error occurs while
logging in`() {
    loginViewRobot.enterEmail(CORRECT_MAIL)
    loginViewRobot.enterPassword(CORRECT_PASSWORD)
    loginViewRobot.clickLogin()

    loginSingleSubject.onError(Throwable("Unknown"))
    overrideSchedulersRule.testScheduler.advanceTimeBy(2, TimeUnit.SECONDS)

    loginViewRobot.enterEmail(CORRECT_MAIL)

    loginViewRobot.assertViewStatesRendered(
        LoginViewState.State(),
        LoginViewState.State(
            progressState = ProgressState.LOGGING_IN
        ),
        LoginViewState.State(
            otherErrors = OtherErrors.UNKNOWN
        ),
        LoginViewState.State()
    )
}
```



```
val viewStateObservable = intent { it.loginClicks }
    .withLatestFrom(emailObservable, passwordObservable) { _, email, password ->
        Pair(email, password)
    }
    .switchMap { (email, password) ->
        val isMailCorrect = loginDataValidator.isCorrectMail(email)
        val isPasswordLongEnough = loginDataValidator
            .isPasswordLongEnough(password)

        if (isMailCorrect && isPasswordLongEnough) {
            login(
                email, password,
                cancelWrongCredentialsObservable(
                    emailObservable, passwordObservable
                )
            )
        } else {
            showPasswordEmailInputError(
                isMailCorrect, isPasswordLongEnough,
                emailObservable, passwordObservable
            )
        }
    }
    .scan<LoginViewState>(LoginViewState.State()) { oldVS, partialChange ->
        when (oldVS) {
            is LoginViewState.State -> reduceState(oldVS, partialChange)
            LoginViewState.Success -> LoginViewState.Success
        }
    }
}
```

```
private fun login(
    email: String, password: String,
    cancelWrongCredentialsErrorObservable:
Observable<PartialStateChange.CancelApiWrongCredentials>
) = loginApi.login(email, password)
    .flatMapCompletable { tokenRepository.saveToken(it) }
    .toSingleDefault<PartialStateChange>(PartialStateChange.Success)
    .toObservable()
    .startWith(PartialStateChange.StartedLoggingIn)
    .onErrorResumeNext { error: Throwable ->
        parseLoginError(error, cancelWrongCredentialsErrorObservable)
    }
}
```

The toast problem

```
private fun timedUnknownErrorObservable(): Observable<PartialStateChange> {  
    return Observable.timer(2, TimeUnit.SECONDS)  
        .map<PartialStateChange> {  
            PartialStateChange.CancelUnknownError  
        }  
        .startWith(PartialStateChange.ApiUnknownError)  
}
```

```
private fun showPasswordEmailInputError(
    isMailCorrect: Boolean, isPasswordLongEnough: Boolean,
    emailObservable: Observable<String>, passwordObservable:
Observable<String>
): Observable<PartialStateChange> = Observable.just<PartialStateChange>(
    PartialStateChange.CredentialsPrevalidationFailed(
        if (isMailCorrect) null else MailInputError.INCORRECT,
        if (isPasswordLongEnough) null else PasswordInputError.TO_SHORT
    )
).concatWith(
    Single.merge(
        emailObservable.firstOnError()
            .map { PartialStateChange.CancelMailInputError },
        passwordObservable.firstOnError()
            .map { PartialStateChange.CancelPasswordInputError }
    ).toObservable()
)
```

Reduce the state


```
private fun reduceState(
    oldState: LoginViewState.State, partialChange: PartialStateChange
) = when (partialChange) {
    is PartialStateChange.CredentialsValidationFailed -> oldState.copy(
        mailInputError = partialChange.mailInputError,
        passwordInputError = partialChange.passwordInputError
    )
    PartialStateChange.CancelMailInputError -> oldState.copy(
        mailInputError = null
    )
    PartialStateChange.CancelPasswordInputError -> oldState.copy(
        passwordInputError = null
    )
    PartialStateChange.StartedLoggingIn -> LoginViewState.State(
        progressState = ProgressState.LOGGING_IN
    )
    PartialStateChange.ApiWrongCredentials -> oldState.copy(
        progressState = ProgressState.WAITS,
        credentialsError = CredentialsError.INCORRECT
    )
    (...)
}
```

```
private fun reduceState(...) = when (partialChange) {  
    (...)  
    PartialStateChange.CancelApiWrongCredentials -> oldState.copy(  
        credentialsError = null  
    )  
    PartialStateChange.ApiUnknownError -> oldState.copy(  
        progressState = ProgressState.WAITS,  
        otherErrors = OtherErrors.UNKNOWN  
    )  
    PartialStateChange.CancelUnknownError -> oldState.copy(  
        otherErrors = null  
    )  
  
    PartialStateChange.Success -> LoginViewState.Success  
}
```

▼	OK	LoginMviViewModelTest (pl.naniewicz.mvisample.feature.login)	185ms
	OK	Should show incorrect credentials when authentication error and clean up if password modified	149ms
	OK	Should show unknown error for 2 seconds if unknown error occurs while logging in	8ms
	OK	Should show incorrect credentials when authentication error and clean up if email modified	0ms
	OK	Should show progressbar and login when correct password	12ms
	OK	Should validate email before making request	15ms
	OK	Should show error when pressing login with empty fields and clear up after modifying given input	1ms



Render!

```
override fun render(viewState: LoginViewState) = when (viewState) {
    LoginViewState.Success -> finishWithSuccess()
    is LoginViewState.State -> renderState(viewState)
}

private fun renderState(viewState: LoginViewState.State) = with(viewState) {
    emailTextInputLayout.error = when (mailInputError) {
        MailInputError.INCORRECT -> getString(...)
        null -> null
    }
    passwordTextInputLayout.error = when (passwordInputError) {
        PasswordInputError.TO_SHORT -> getString(...)
        null -> null
    }
    credentialsErrorText.text = when (credentialsError) {
        CredentialsError.INCORRECT -> getString(...)
        null -> null
    }
    when (otherErrors) {
        OtherErrors.UNKNOWN -> showUnknownError()
        null -> hideUnknownError()
    }
    progressBar.visibility = when (progressState) {
        LOGGING_IN -> View.VISIBLE
        WAITS -> View.GONE
    }
}
```

MVI good practices

Do:

- Export state reducer's - when your state reducing is growing uncontrollably - it might be also worth to unit test such class in isolation.
- Export interactors - they can expose wrap(...) methods for passing intent's and retrieving observable
- Create smaller self manageable parts - mvi view model for view's, fragments.
- Use KOTLIN
- Create „nice” view state's

„Nice” view state

- Remember that it's view state responsibility to represent view state. It shouldn't be simply mutating your View/Fragment/Activity.
- Representing view states as Enum's / Sealed classes often leads to placing additional logic into render functions that can't be tested, use responsibly.

MVI pros and cons

- + Clear data flow
- + Great testability
- + Makes you stop thinking about configuration changes
- + Dumb views
- + Unleashing the full Rx superpowers
- + Pr's: Woah dude, that's cool
- Boilerplate
- Everything needs to be reactive
- Sometimes we might have to work against framework / libraries
- High learning cost
- Pr's: Woah dude what is going on there



Sample and presentation available here:
<https://github.com/freszu/MVI-Sample>

Rafał Naniewicz
Android Developer @Netguru
rafal.naniewicz@netguru.co
rafal.naniewicz@gmail.com

