Testing By Design

Kostia Tarasenko

Hannes Dorfmann

FREELETICS

FREELETICS

6 YEARS OF GROWTH AND COUNTING

YEAR
2013

FREE ATHLETES IN THE WORLD
390,863

How to write maintainable tests and

how proper architecture can help you

Unit tests

White box testing of individual software components in isolation

Integration tests

“Narrow” definition:

Testing the code that connects

different components

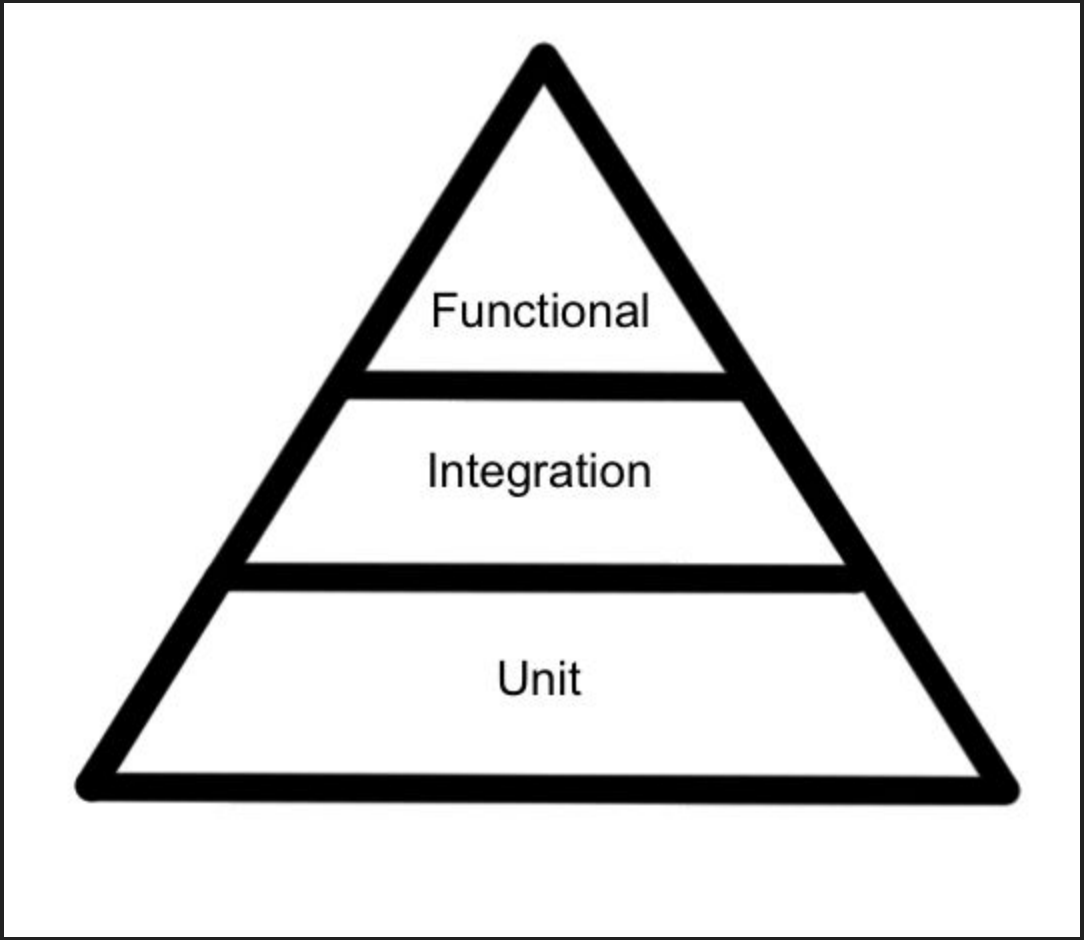
Functional tests

Black box testing.

Feeding input and examining the output

End-to-end tests

Functional tests backed by real environment


Functional
Integration
Unit

Why bother?

Functional tests cover

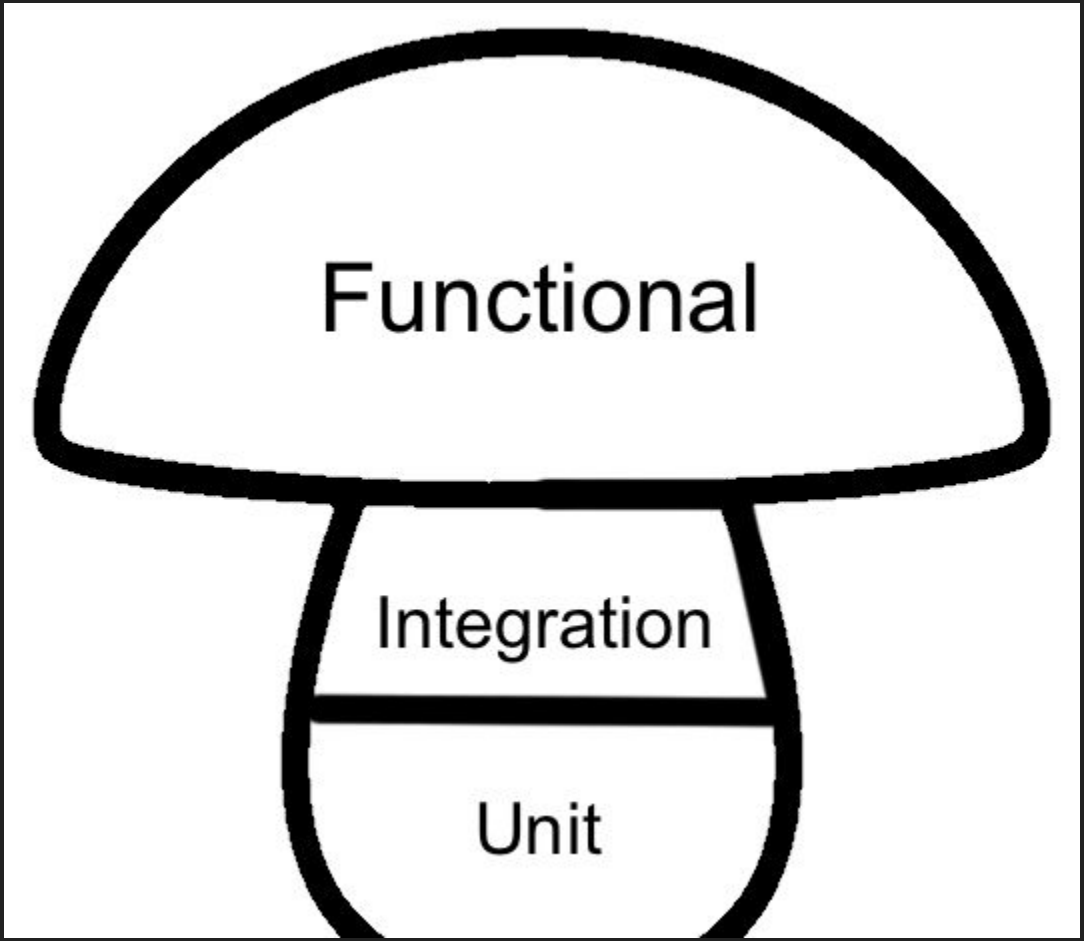
integration and unit tests scope

Challenges

- Isolation of side effects
- Unreliable slow layers: network, disk access, db access
- Every code change affects tests

What if?

… writing and maintaining functional tests was easier?


Functional
Integration
Unit

Uncle Bob says:

Keep the test code to the same level of quality as the production code.

Test code is not throw-away code.

```
@Test
    fun newTodoItemIsShownInTodoList() {
        onView(withId(R.id.newItem)).perform(click())
        onView(withId(R.id.step1Title)).perform(typeText("Another Item"))
        onView(withId(R.id.button)).perform(click())
        onView(withId(R.id.create)).perform(click())
        onView(withText("Another Item")).check(matches(isDisplayed()))
    }
```

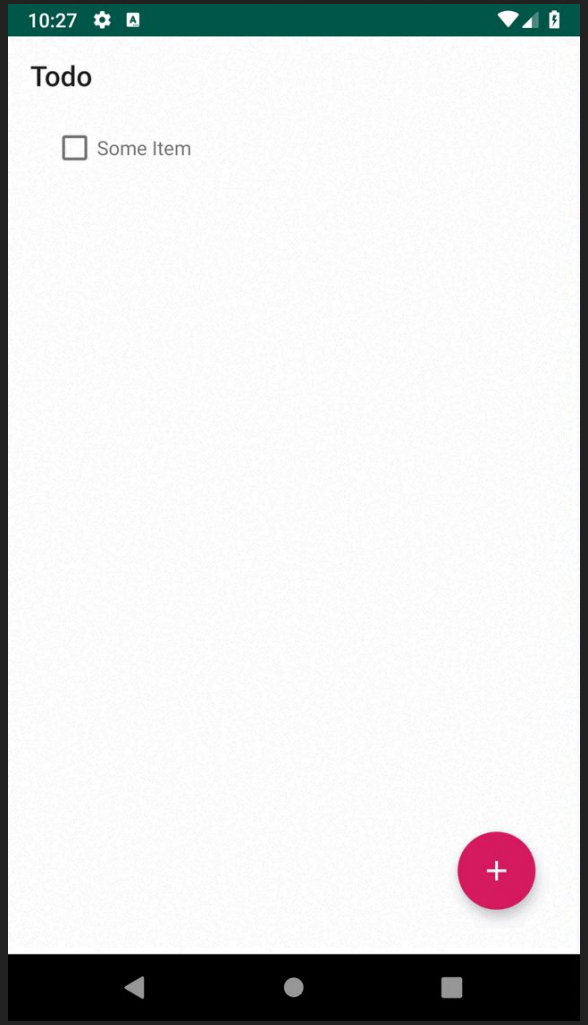
Write tests yo mama would be proud of

Robot pattern

- influenced by Martin Fowler’s PageObject pattern
- Embraced by kotlin DSL syntax

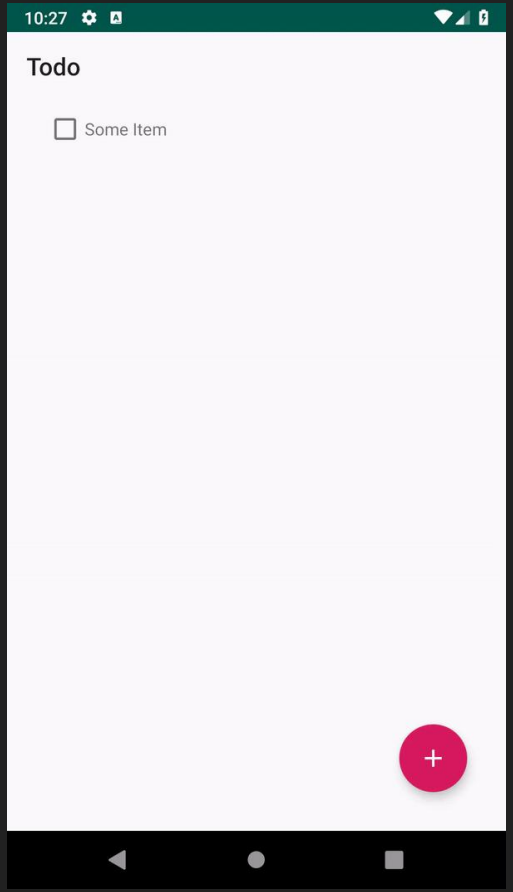
Robot pattern. Key principles

- Let robot do what user can do
- Verify what user would see

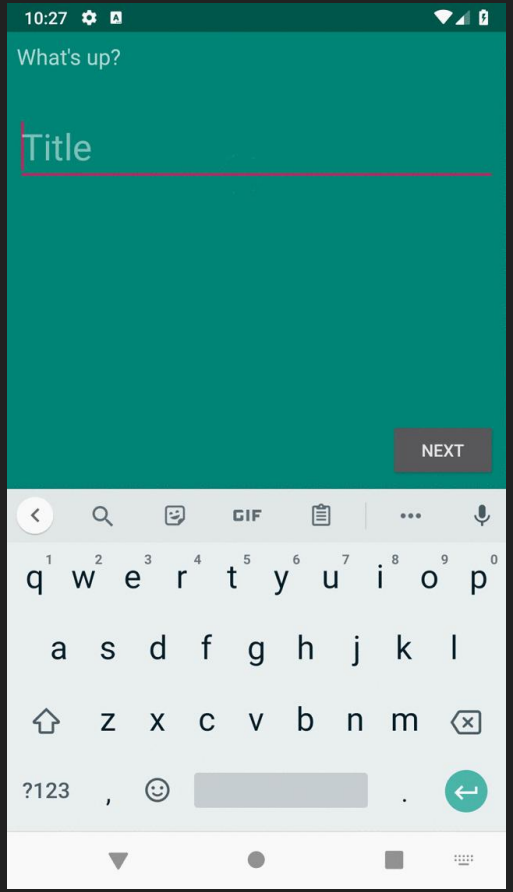

10:27
Todo
Some Item
+

```
@Test
fun newTodoItemIsShownInTodoList() {




}
```

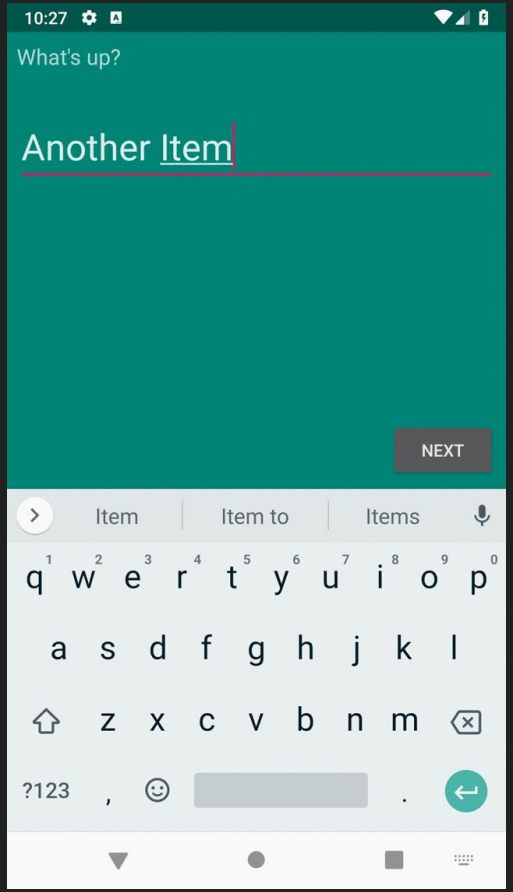
```
@Test
fun newTodoItemIsShownInTodoList() {
    todoList {
            clickCreateTodoItem()




    }
}
```



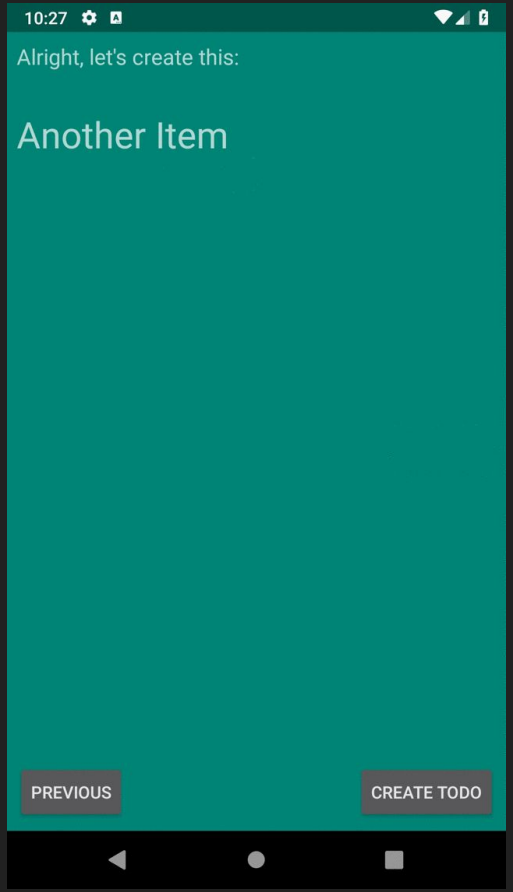
```
@Test
fun newTodoItemIsShownInTodoList() {
    todoList {
            clickCreateTodoItem()

            createItem {



            }
    }
}
```



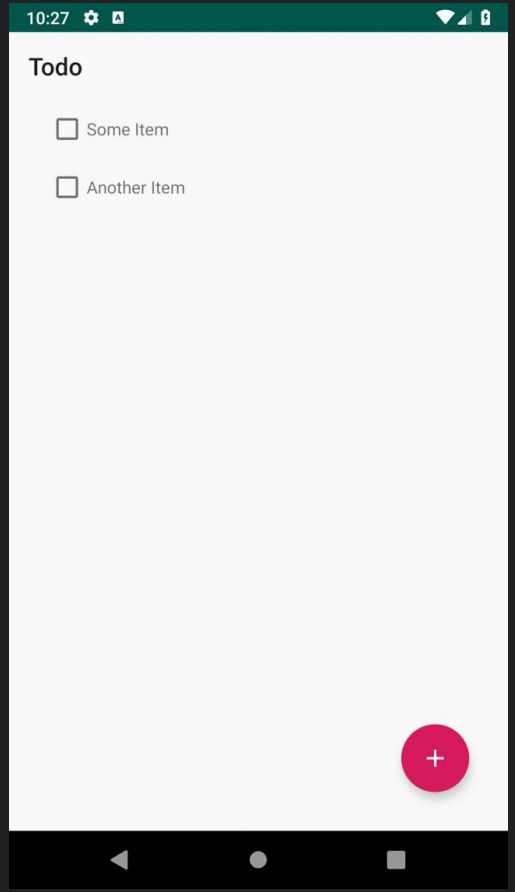
```
@Test
fun newTodoItemIsShownInTodoList() {
    todoList {
            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")


            }
        }
}
```



```
@Test
fun newTodoItemIsShownInTodoList() {
    todoList {
            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                pressNext()

            }
        }
}
```



```
@Test
fun newTodoItemIsShownInTodoList() {
    todoList {
            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                pressNext()
                pressSave()
            }
        }
}
```



Robot pattern. Advantages

- No need to change tests if the flow doesn’t change
- Fun to read
- Easy to extend

Problems with “traditional” Robot Pattern

- In which state is the Robot starting?
- Assertions hidden
- Doesn’t read like a specification

```kotlin
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)


        todoList {









        }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {


        todoList {









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState()









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState

        }
    }
```

```
val assertLoadingState: Unit
    get() = ...
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()









        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {









            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")









            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState









            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()









            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState



            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()



            }



        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState

            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }


        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

```
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled









    }
}
```

```
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

        todoList {
            assertLoadingState
            assertContentState + prefilled









        }
    }
}
```

```
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

        todoList {
            assertLoadingState
            assertContentState + prefilled

            clickFirstItem()









        }
    }
}
```

```
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

        todoList {
            assertLoadingState
            assertContentState + prefilled

            clickFirstItem()

            assertContentStateWithFirstItemDone




        }
    }
}
```

```kotlin
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

        todoList {
            assertLoadingState
            assertContentState + prefilled

            clickFirstItem()

            assertContentStateWithFirstItemDone

            clickFirstItem()


        }
    }
}
```

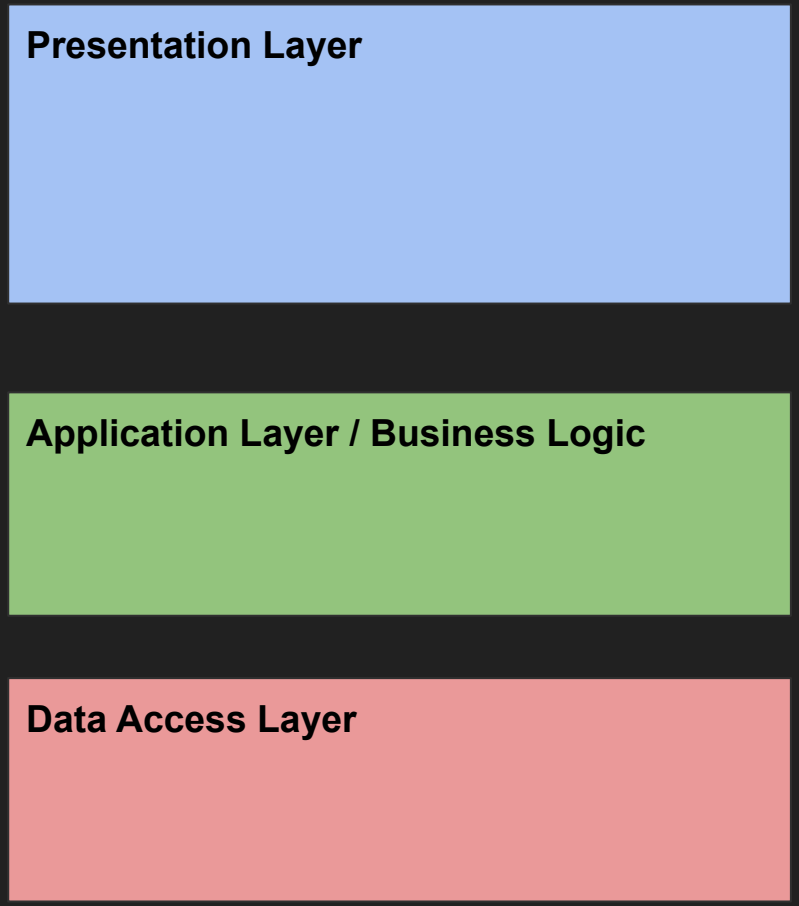
```kotlin
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

        todoList {
            assertLoadingState
            assertContentState + prefilled

            clickFirstItem()

            assertContentStateWithFirstItemDone

            clickFirstItem()

            assertContentStateWithFirstItemNotDone
        }
    }
}
```

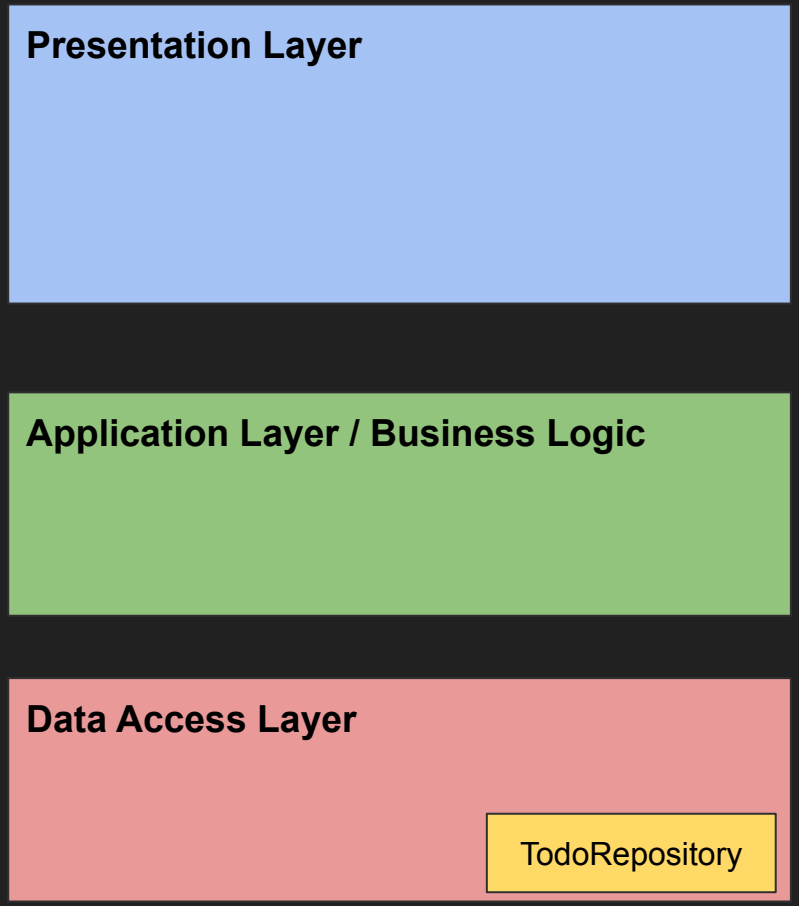
```
@Test
fun navigateBackAndForthInCreateItemWizard() {

    given {
        initialState = SummaryState("Some item")

        createItem {
            assertSummaryState

            pressBack()

            assertEnterTitleState

            pressNext()

            assertSummaryState
        }
    }
}
```

but …. HOW?

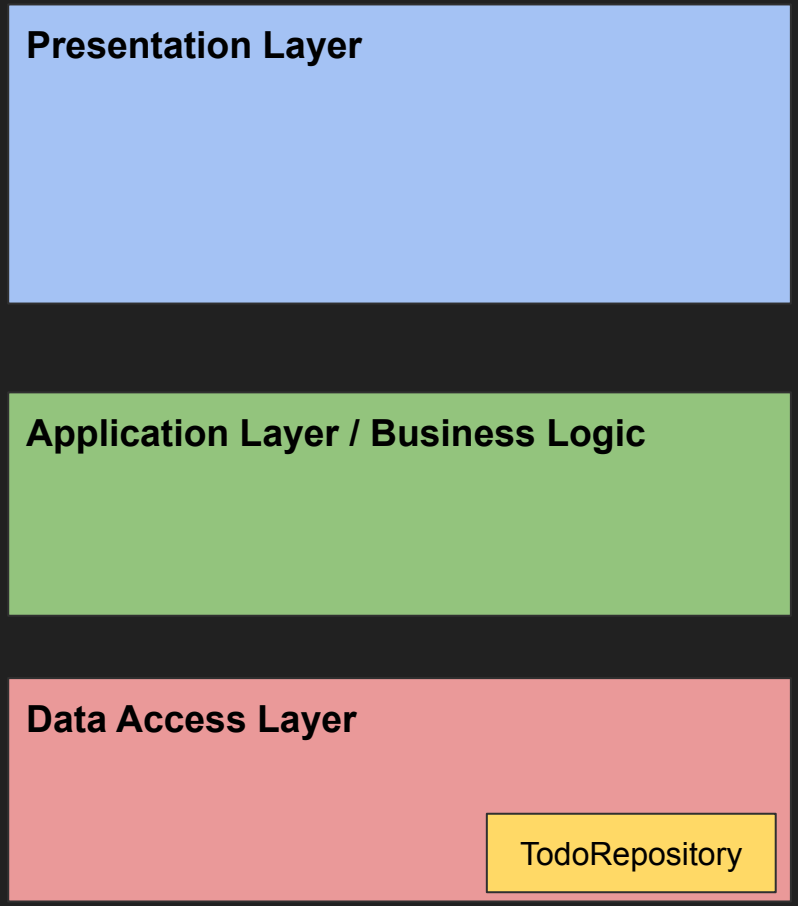
State Based Architecture

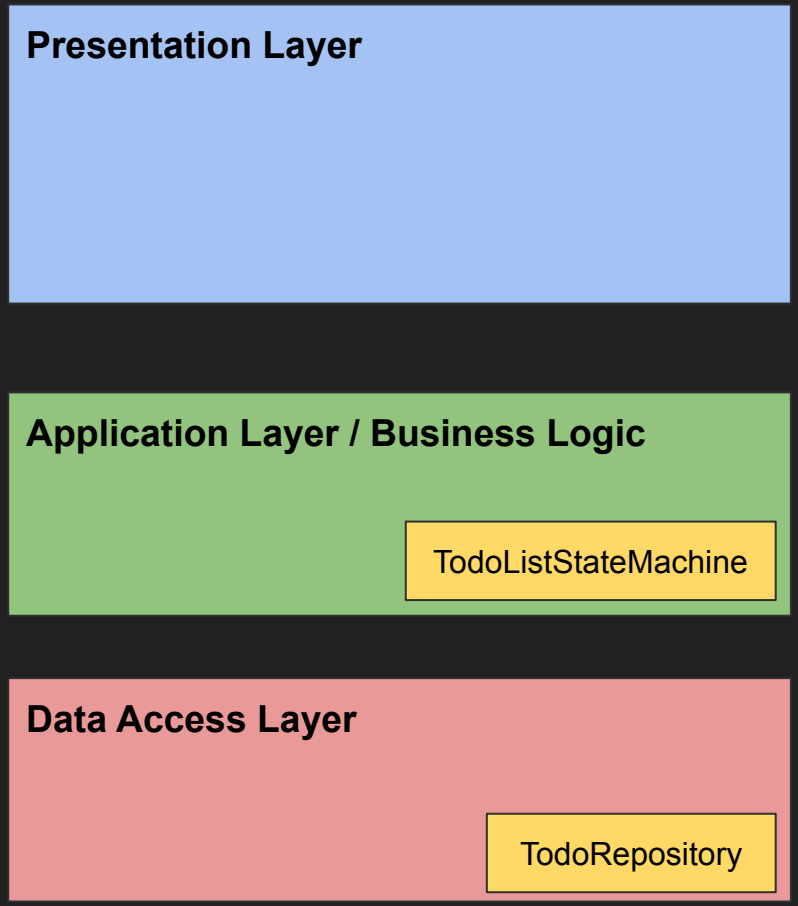
- Business Logic based on state machines
- Single source of truth
- State gets “rendered” on screen
- Atomic UI updates
- Push, not pull


Presentation Layer
Application Layer / Business Logic
Data Access Layer


Presentation Layer
Application Layer / Business Logic
Data Access Layer
TodoRepository

```
interface TodoRepository {
    fun getAll() : Observable<List<TodoItem>>
    fun add(item : TodoItem)
    fun update(item : TodoItem)
}
```


Presentation Layer
Application Layer / Business Logic
Data Access Layer
TodoRepository


Presentation Layer
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository

```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    val state: Observable<State>

    fun input(action: Action) {

    }
}
```

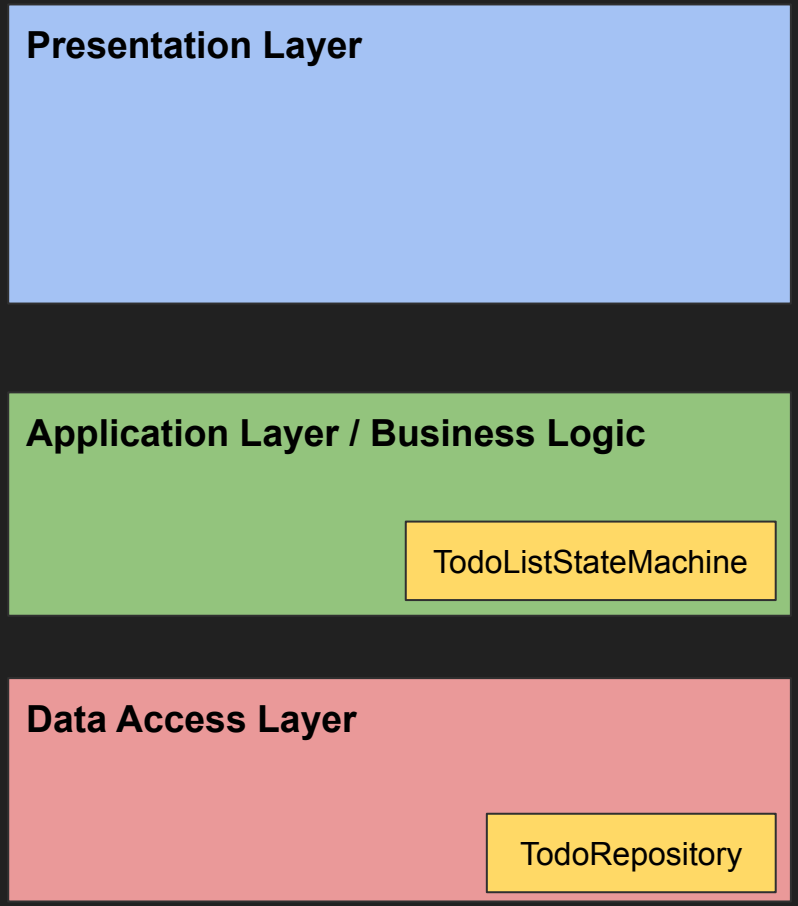
```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    sealed class State {
        object Loading : State()
        data class Content(val items: List<TodoItem>) : State()
        object Error : State()
    }




    val state: Observable<State>




    fun input(action: Action) {




    }
}
```

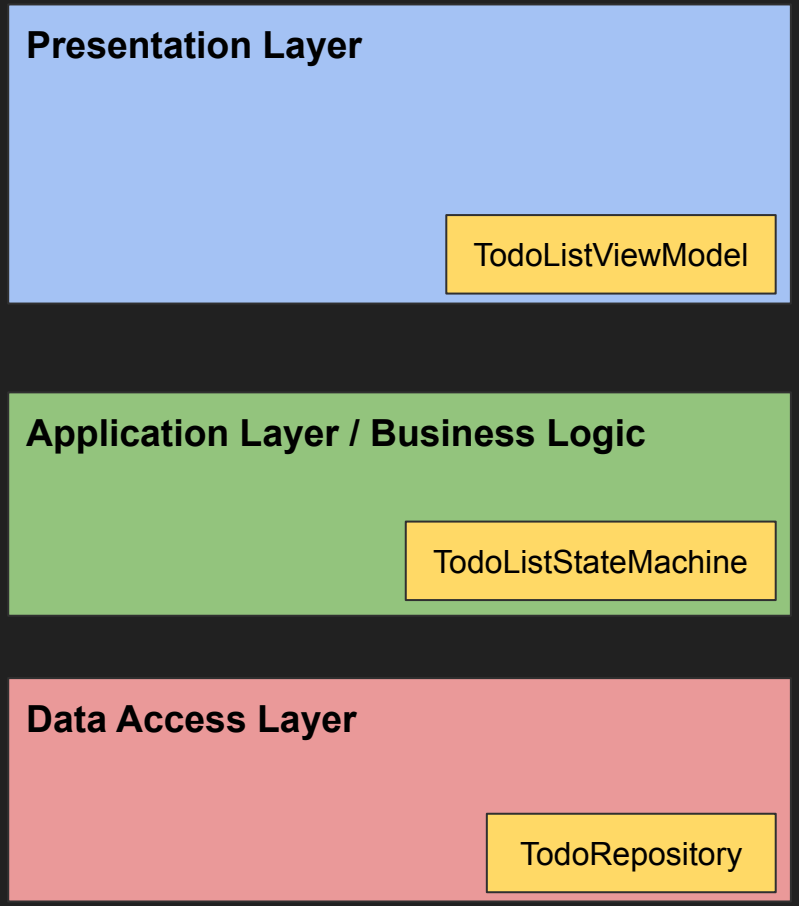
```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    sealed class State {
        object Loading : State()
        data class Content(val items: List<TodoItem>) : State()
        object Error : State()
    }

    val state: Observable<State> =
        repository.getAll()
            .map { State.Content(it) as State }
            .onErrorReturn { State.Error }
            .startWith(State.Loading)

    fun input(action: Action) {

    }
}
```

```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    sealed class State {
        object Loading : State()
        data class Content(val items: List<TodoItem>) : State()
        object Error : State()
    }




    val state: Observable<State> = ...




    fun input(action: Action) {




    }
}
```

```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    sealed class State {
        object Loading : State()
        data class Content(val items: List<TodoItem>) : State()
        object Error : State()
    }

    sealed class Action {
        data class ToggleTodoItemDoneAction(val item: TodoItem) : Action()
        data class DeleteTodoItemAction(val item: TodoItem) : Action()
    }

    val state: Observable<State> = ...




    fun input(action: Action) {




    }
}
```

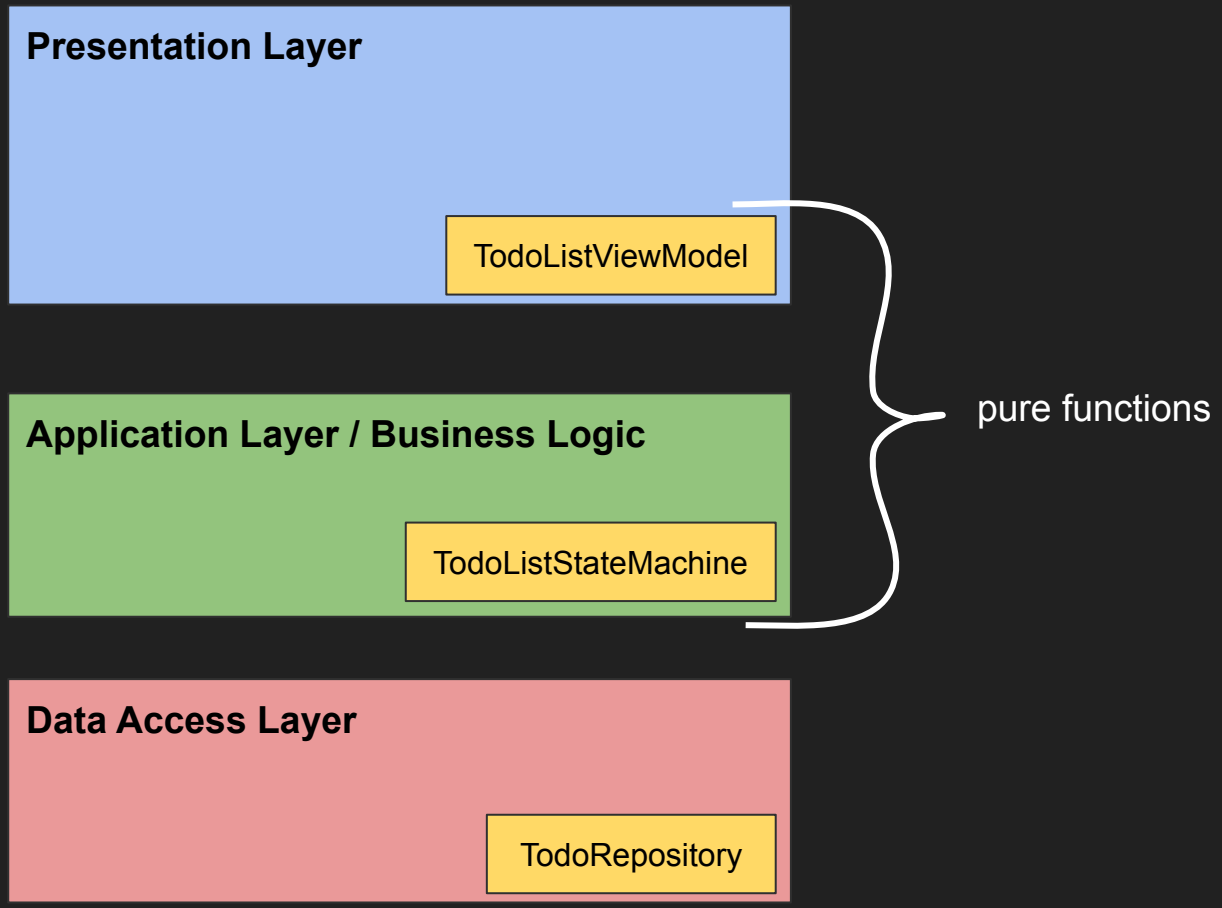
```
class TodoListStateMachine @Inject constructor(private val repository: TodoRepository) {

    sealed class State {
        object Loading : State()
        data class Content(val items: List<TodoItem>) : State()
        object Error : State()
    }

    sealed class Action {
        data class ToggleTodoItemDoneAction(val item: TodoItem) : Action()
        data class DeleteTodoItemAction(val item: TodoItem) : Action()
    }

    val state: Observable<State> = ...




    fun input(action: Action) {
        when (action) {

            ...

        }
    }
}
```

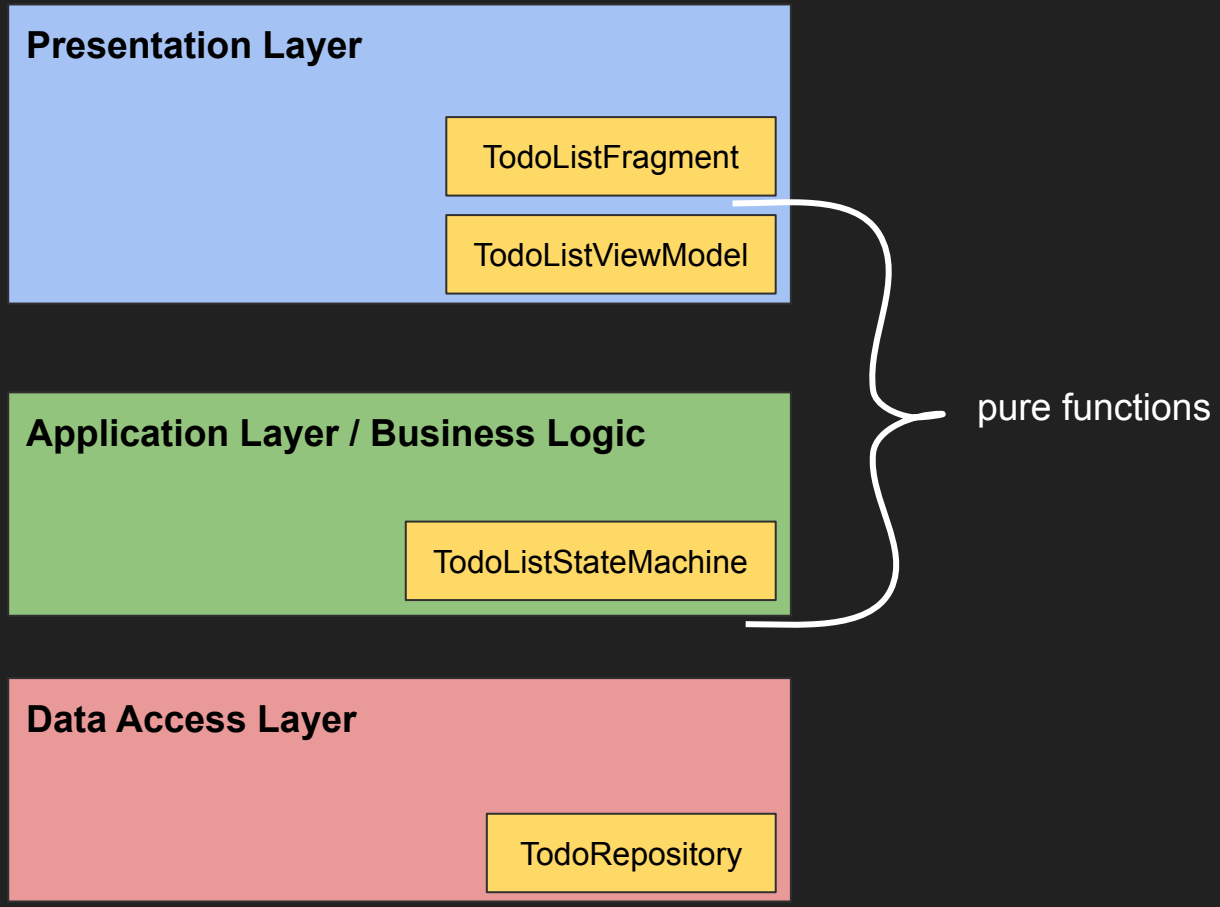

Presentation Layer
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository


Presentation Layer
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository

```
class TodoListViewModel @Inject constructor(
    private val stateMachine: TodoListStateMachine
) : ViewModel() {




    val state = MutableLiveData<State>()

    fun input(action: Action) {

    }



}
```

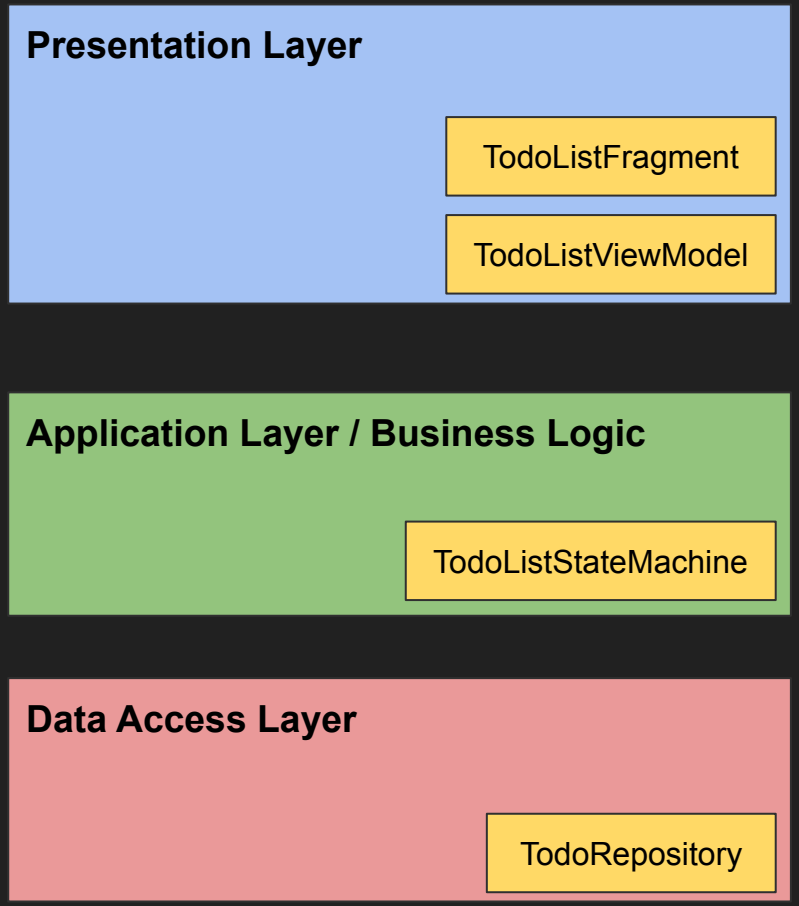
```
class TodoListViewModel @Inject constructor(
    private val stateMachine: TodoListStateMachine
) : ViewModel() {

    private val disposable: Disposable = stateMachine.state
                                            .subscribeOn(Schedulers.io())
                                            .subscribe { state.postValue(it) },


    val state = MutableLiveData<State>()

    fun input(action: Action) {
        stateMachine.input(action)
    }

    override fun onCleared() {
        disposable.dispose()
    }
}
```

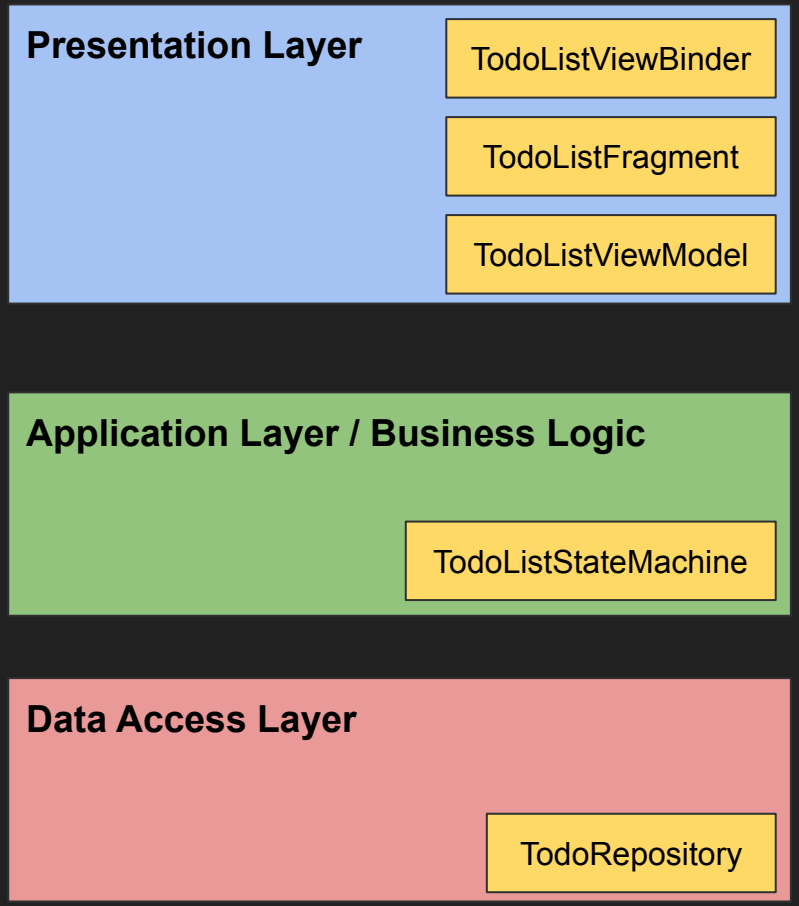

Presentation Layer
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions


Presentation Layer
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions

```
class TodoListViewFragment : Fragment() {


    @Inject
    lateinit var viewModel : TodoListViewModel

    @Inject
    lateinit var viewBinder: TodoListViewBinder

    override fun onCreateView(inflater: LayoutInflater, c: ViewGroup?, b: Bundle?): View? =
        inflater.inflate(R.layout.fragment_todolist, c, false)

    override fun onStart() {
        super.onStart()
        viewModel.state.observe(this, Observer { viewBinder.render(it) })
        viewBinder.actionListener = viewModel::input
    }
}
```

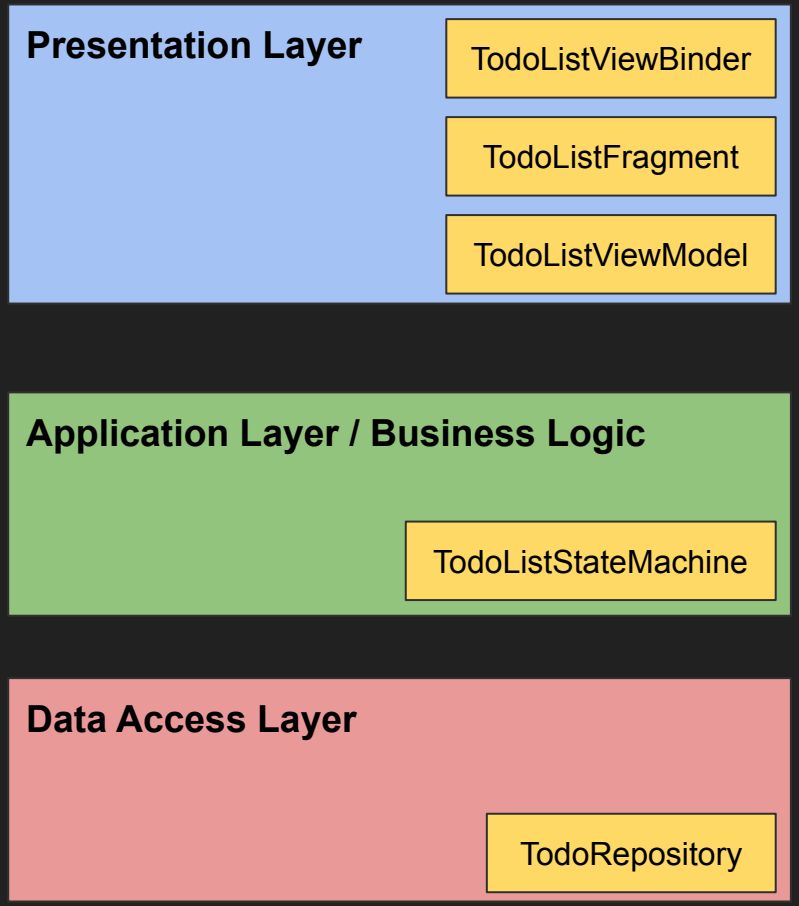
```
class TodoListViewFragment : Fragment() {


    @Inject
    lateinit var viewModel : TodoListViewModel

    @Inject
    lateinit var viewBinder: TodoListViewBinder

    override fun onCreateView(inflater: LayoutInflater, c: ViewGroup?, b: Bundle?): View? =
        inflater.inflate(R.layout.fragment_todolist, container, false)

    override fun onStart() {
        super.onStart()
        viewModel.state.observe(this, Observer { viewBinder.render(it) })
        viewBinder.actionListener = viewModel::input
    }
}
```

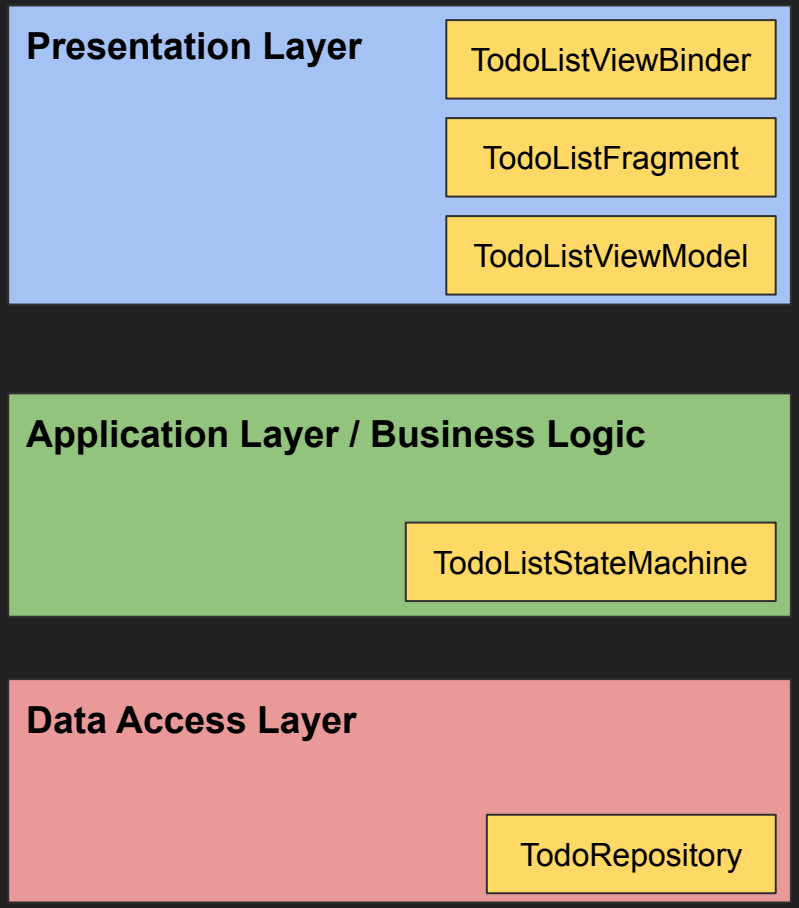

Presentation Layer
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository

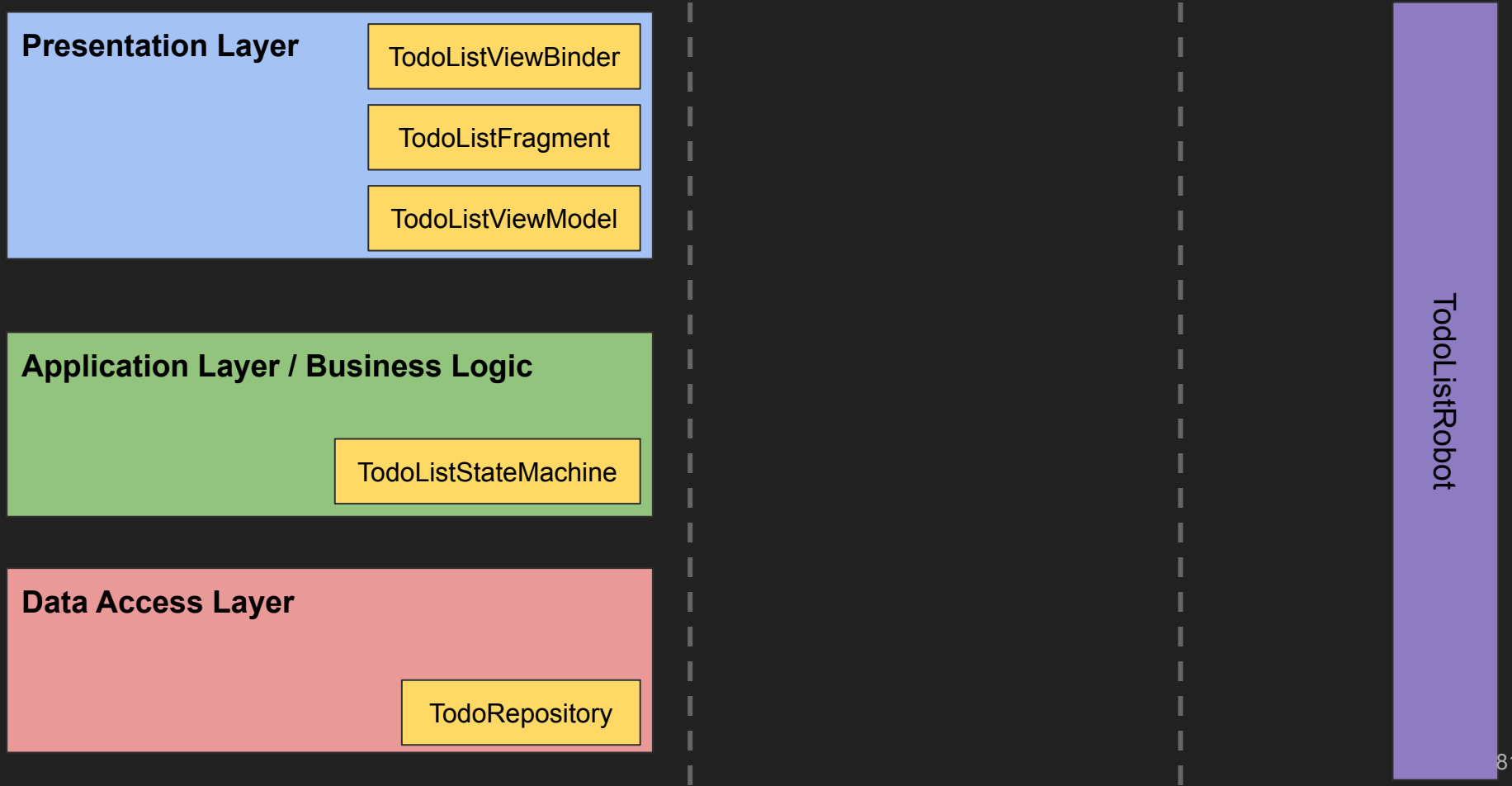

Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository

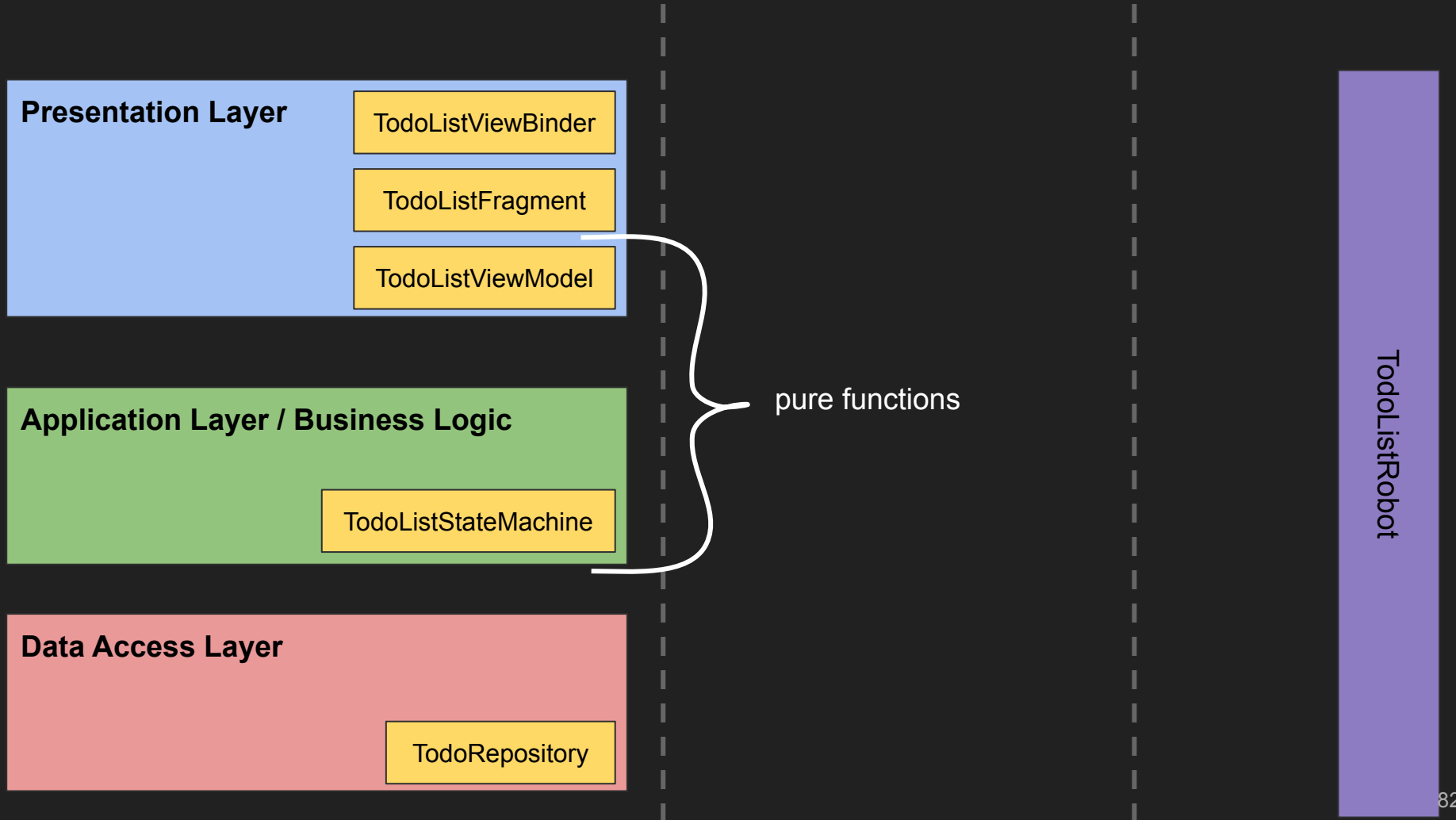
```
open class TodoListViewBinder(private val root: View) {

    lateinit var actionListener: (Action) -> Unit
    private val adapter = TodoListAdapter({ actionListener(it) })
    private val recyclerView = root.findViewById<RecyclerView>(R.id.recyclerView)
    private val error = root.findViewById<View>(R.id.error)
    private val loading = root.findViewById<View>(R.id.loading)

    open fun render(state: State) = when (state) {
        TodoListStateMachine.State.Loading -> {
            loading.visible()
            ...
        }
        TodoListStateMachine.State.Error -> {
            error.visible()
            ...
        }
        is TodoListStateMachine.State.Content -> {
            recyclerView.visible()
            adapter.items = state.items
            adapter.notifyDataSetChanged()
            ...
        }
    }
}
```

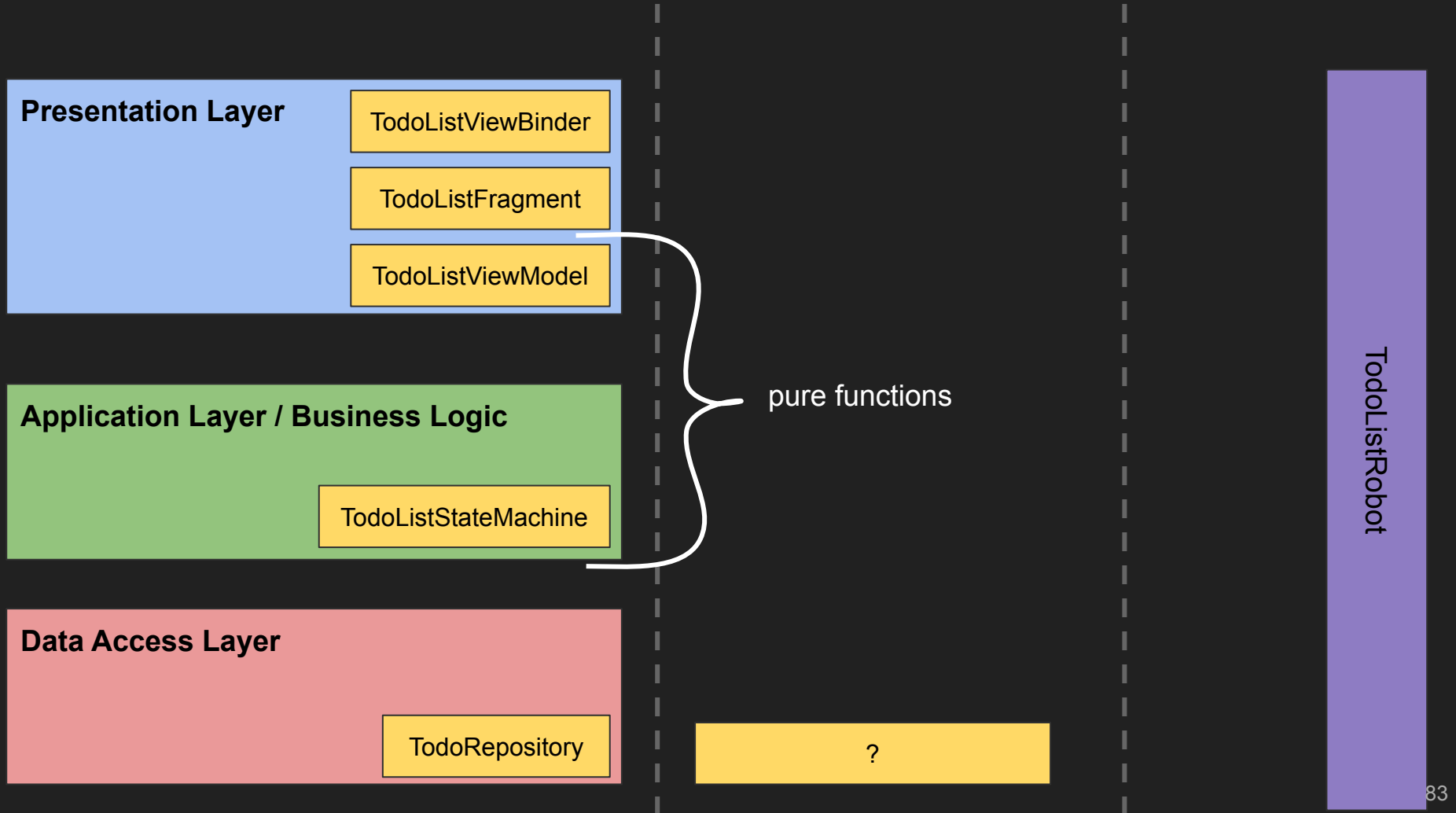
```
open class TodoListViewBinder(private val root: View) {

    lateinit var actionListener: (Action) -> Unit
    private val adapter = TodoListAdapter({ actionListener(it) })
    private val recyclerView = root.findViewById<RecyclerView>(R.id.recyclerView)
    private val error = root.findViewById<View>(R.id.error)
    private val loading = root.findViewById<View>(R.id.loading)

    open fun render(state: State) = when (state) {
        TodoListStateMachine.State.Loading -> {
            loading.visible()
            ...
        }
        TodoListStateMachine.State.Error -> {
            error.visible()
            ...
        }
        is TodoListStateMachine.State.Content -> {
            recyclerView.visible()
            adapter.items = state.items
            adapter.notifyDataSetChanged()
            ...
        }
    }
}
```


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
TodoListRobot


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
TodoListRobot

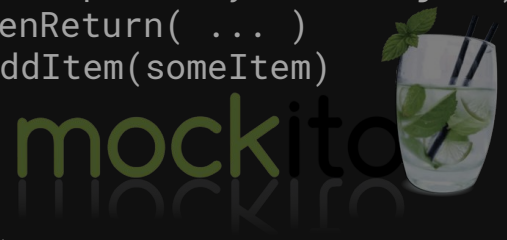

Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
?
TodoListRobot


mockito


```kotlin
@Test
fun test1() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}
```


```
@Test
fun test1() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}

@Test
fun test2() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}
```

```kotlin
@Test
fun test3() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}
```



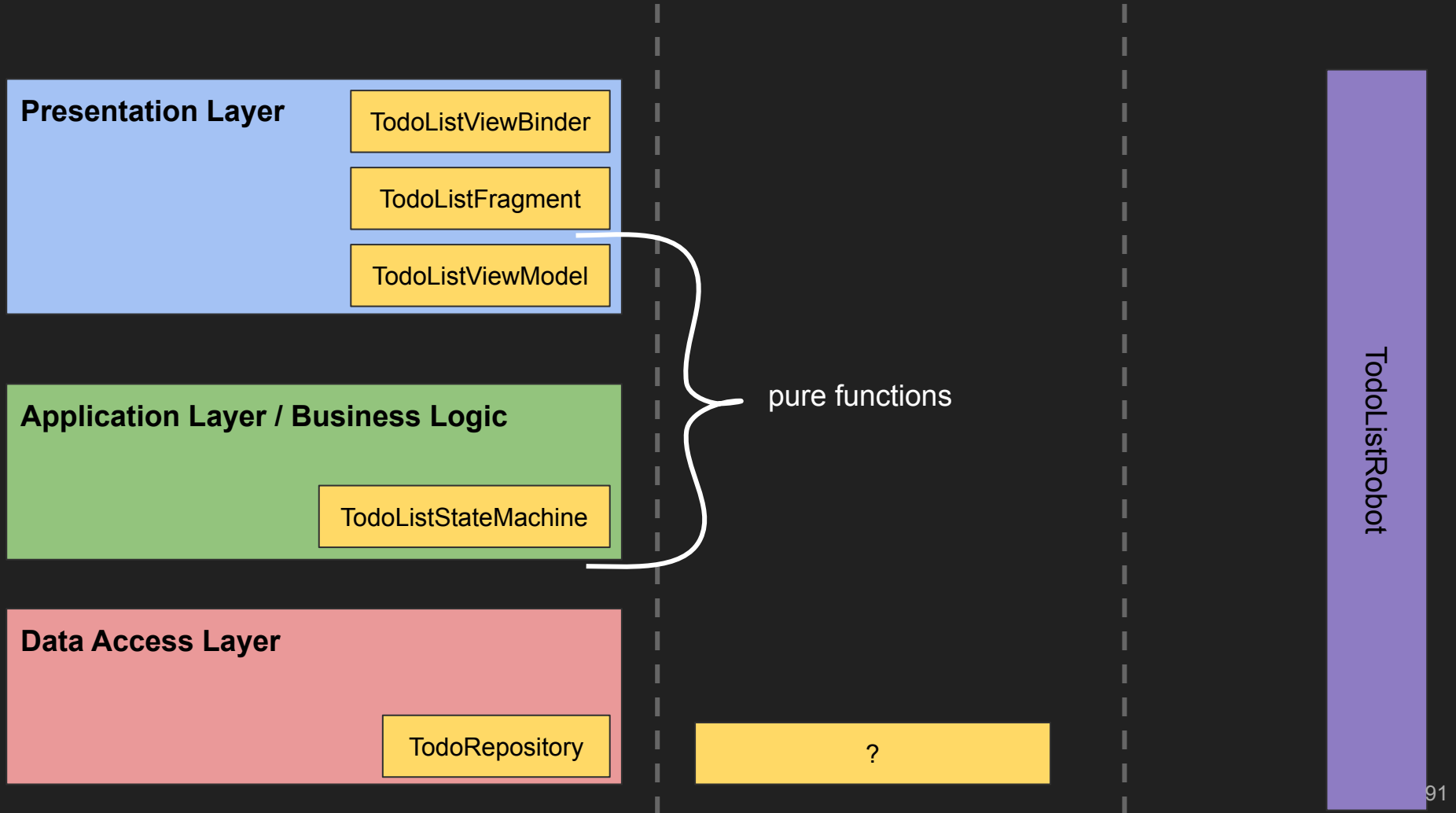

```kotlin
@Test
fun test4() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}

@Test
fun test5() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}

@Test
fun test6() {
  val repository = Mockito.mock(TodoRepository::class.java)
  `when`( repository.getAll() ).thenReturn( ... )
  verify( repository, times(1) ).addItem(someItem)
}

@Test
fun test7() {
  val repository = Mockito.mock
  `when`( repository.getAll() )
  verify( repository, times(1)
}

@Test
fun test8() {
  val repository = Mockito.mo
  `when`( repository.getAll()
  verify( repository, times(1
}

@Test
fun test9() {
  val repository = Mockito.
  `when`( repository.getAll
  verify( repository, times
}
```

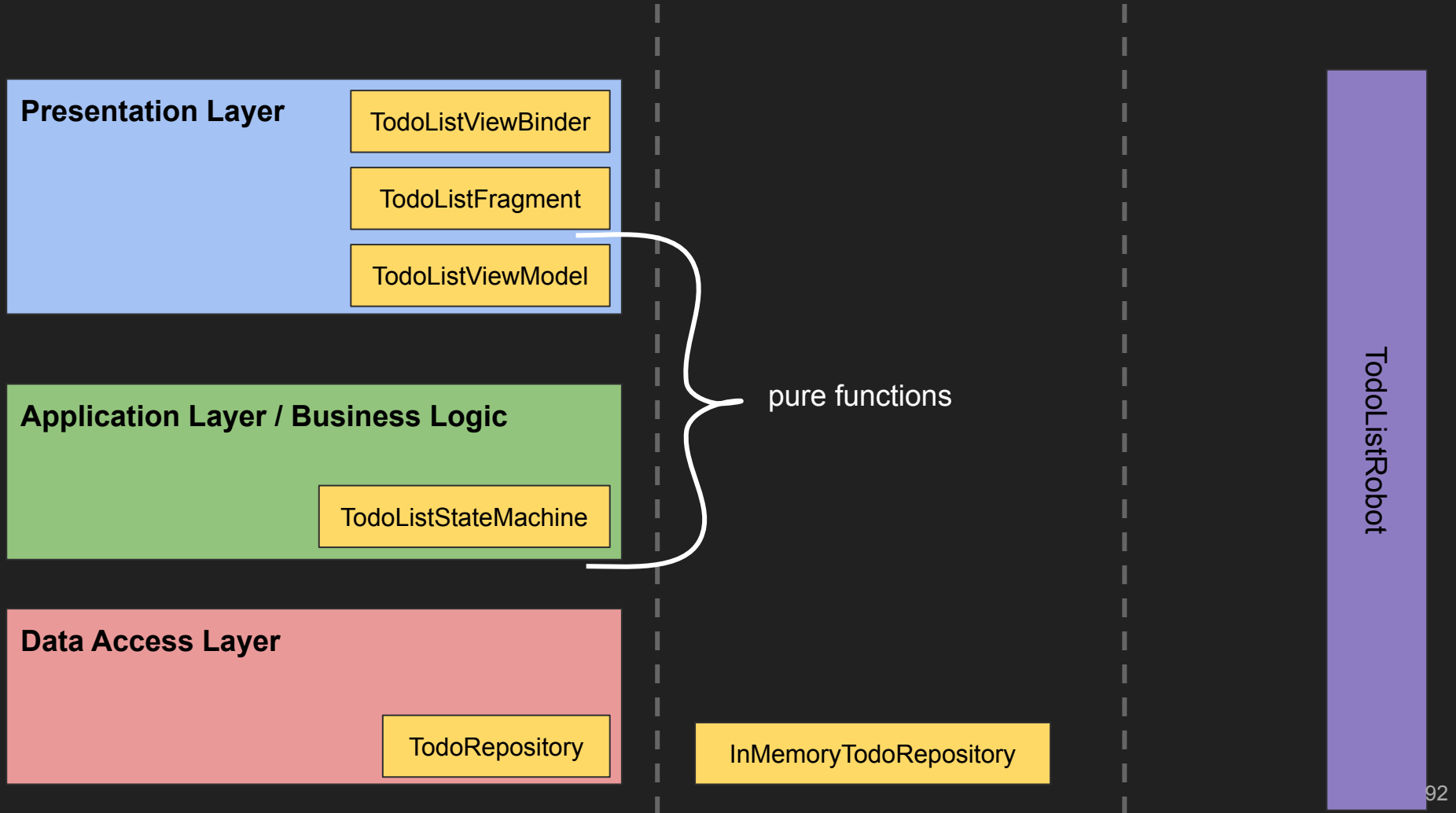

mockito

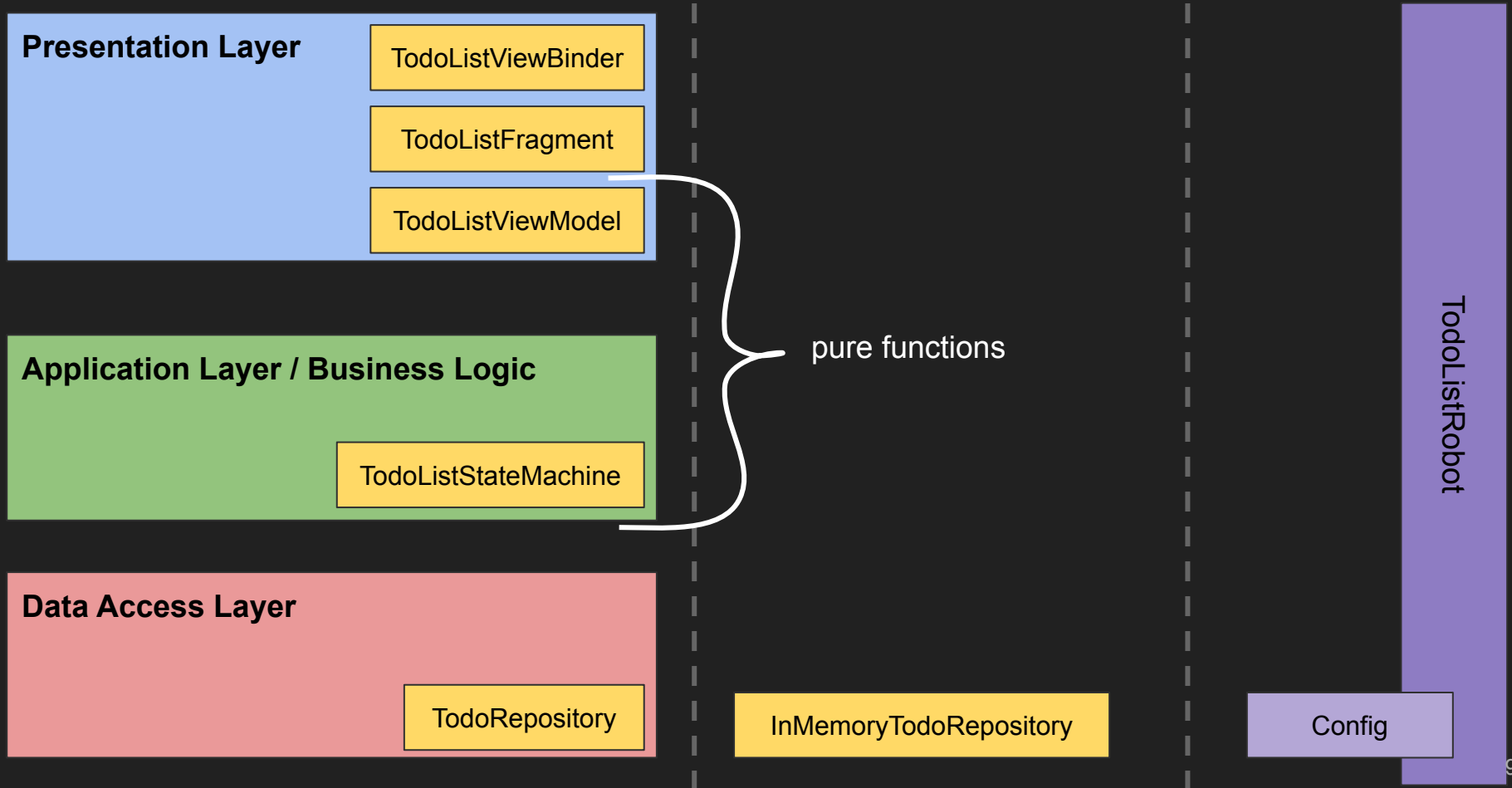

mockito

write fakes / test implementations instead

Refactoring
Reusable


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
?
TodoListRobot


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
InMemoryTodoRepository
TodoListRobot


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
InMemoryTodoRepository
Config
TodoListRobot

```
@Test
fun markAsDoneAndNotDone() {
    val prefilled = listOf(
        TodoItem("1", "First Item", false),
        TodoItem("2", "Second item", true) )

    given {
        prefilledTodoItems = prefilled

         ...

         todoList {
            ...
         }

     }



}
```

```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}
```

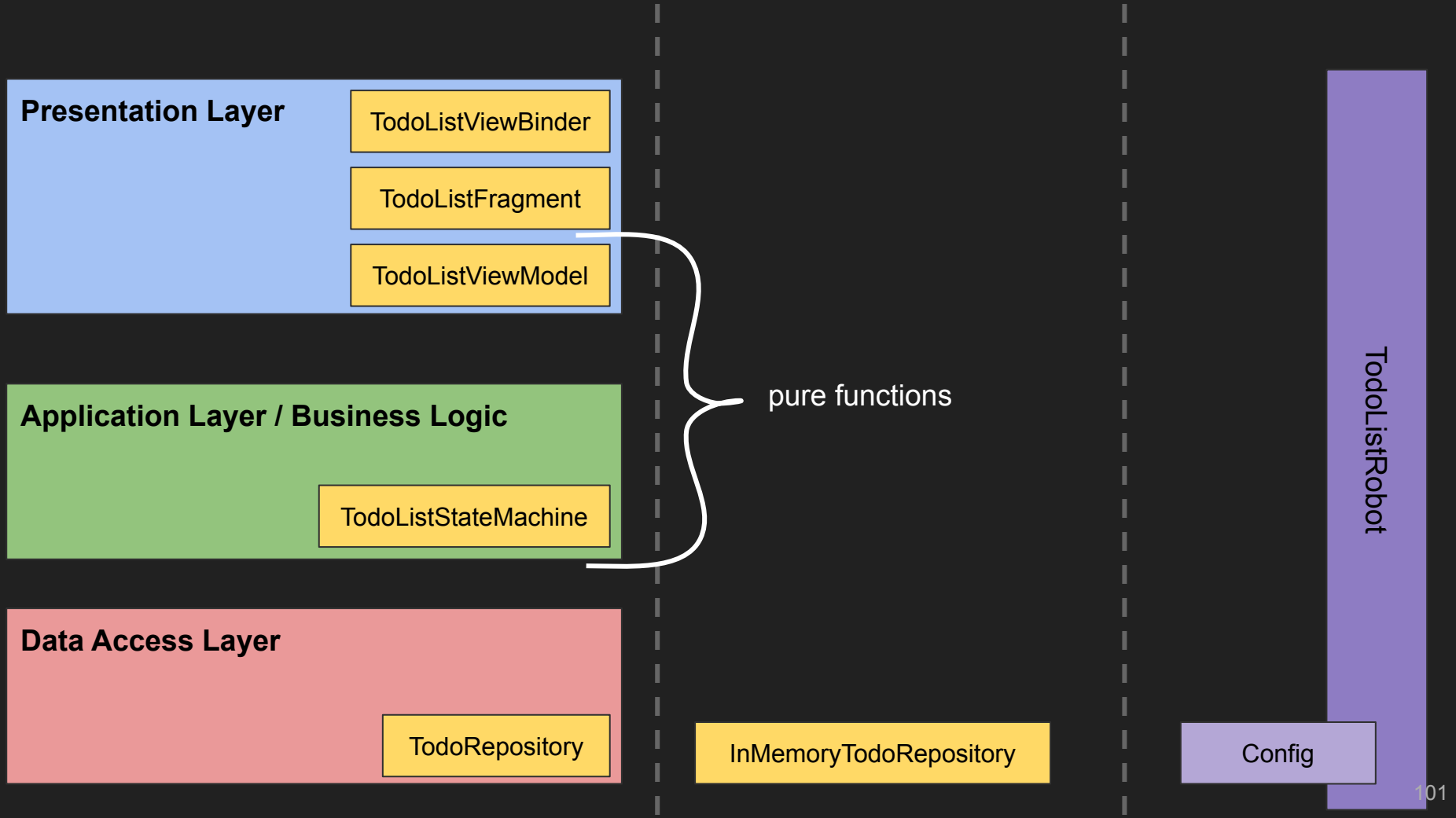
```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}


class Config {

   var prefilledTodoItems = emptyList<TodoItem>()









}
```

```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}


class Config {

   var prefilledTodoItems = emptyList<TodoItem>()

   fun todoList(block: TodoListRobot.() -> Unit) {









   }
}
```

```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}


class Config {

   var prefilledTodoItems = emptyList<TodoItem>()

   fun todoList(block: TodoListRobot.() -> Unit) {
       val todoRepository : InMemoryTodoRepository = ...









   }
}
```

```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}


class Config {

   var prefilledTodoItems = emptyList<TodoItem>()

   fun todoList(block: TodoListRobot.() -> Unit) {
       val todoRepository : InMemoryTodoRepository = ...

       todoRepository.clear()
       prefilledTodoItems.forEach {
           todoRepository.add(it)
       }



   }
}
```

```
fun given(configBlock: Config.() -> Unit) {
      val config = Config(activityRule)
      configBlock(config)
}


class Config {

   var prefilledTodoItems = emptyList<TodoItem>()

   fun todoList(block: TodoListRobot.() -> Unit) {
       val todoRepository : InMemoryTodoRepository = ...

       todoRepository.clear()
       prefilledTodoItems.forEach {
           todoRepository.add(it)
       }

       val robot = TodoListRobot()
       block(robot)
   }
}
```

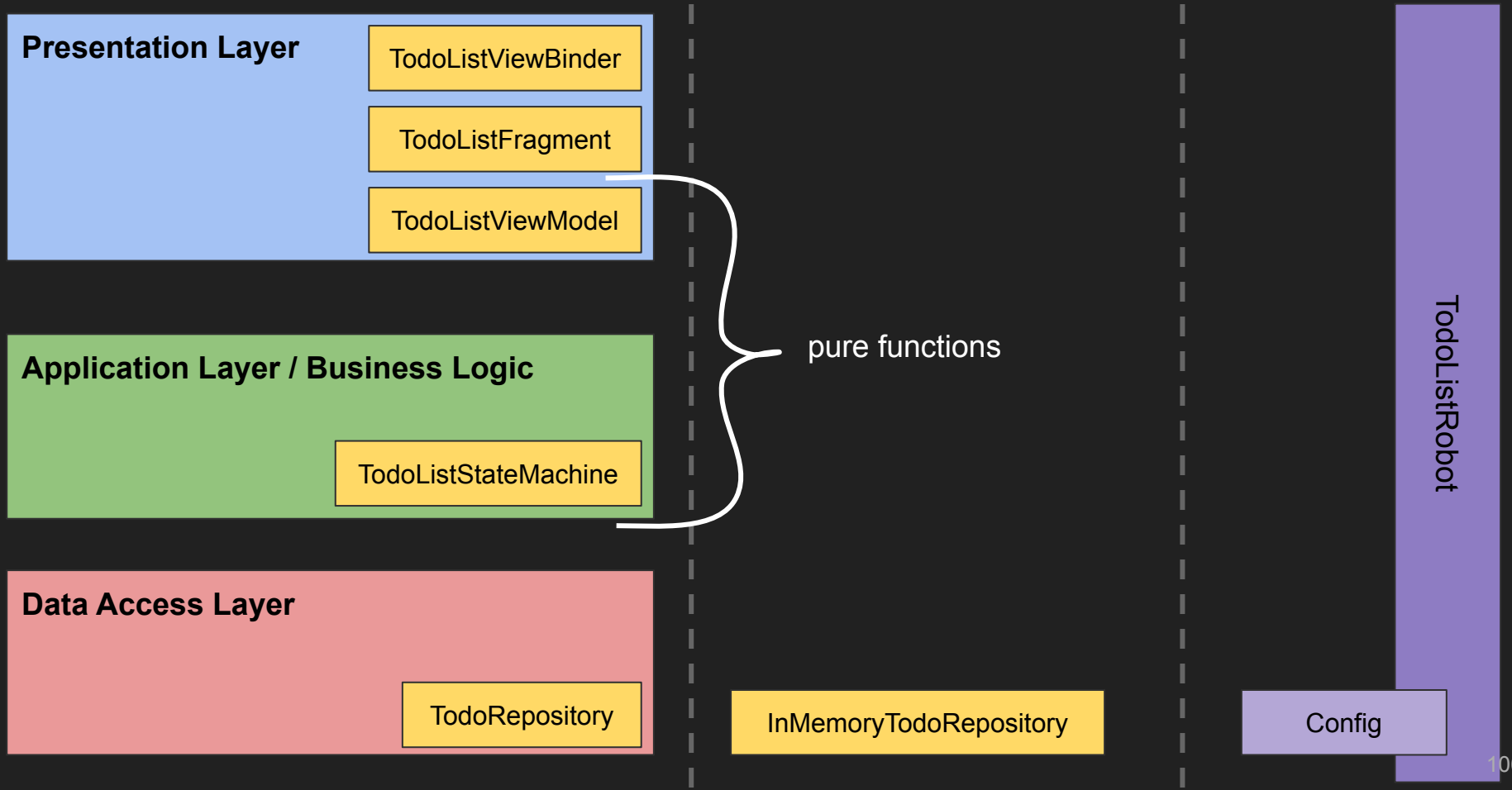

Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
InMemoryTodoRepository
Config
TodoListRobot

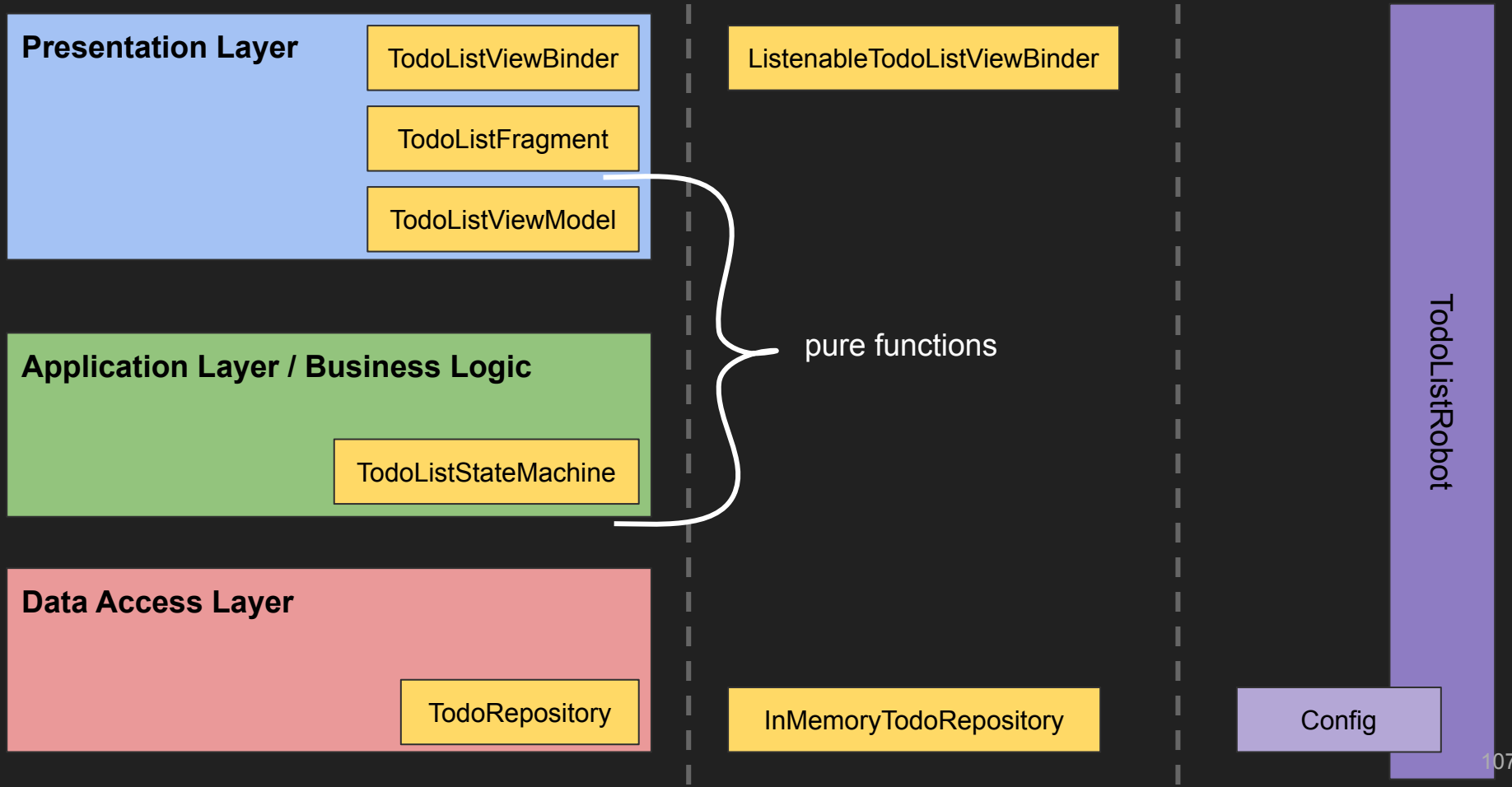
```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

```
class TodoListRobot {




}
```

```
class TodoListRobot {

    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
InMemoryTodoRepository
Config
TodoListRobot


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
ListenableTodoListViewBinder
InMemoryTodoRepository
Config
TodoListRobot

```
class ListenableTodoListViewBinder(root: View) : TodoListViewBinder(root) {

    lateinit var stateChangedListener: (State) -> Unit

    override fun render(state: TodoListStateMachine.State) {
        super.render(state)
        stateChangedListener(state)
    }

}
```

```
class TodoListRobot {

    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    init {
        val viewBinder : ListenableTodoListViewBinder = ...
        viewBinder.stateChangedListener =  { state ->   ...                    }
    }

    fun clickCreateTodoItem() {
        Espresso.onView(ViewMatchers.withId(R.id.newItem))
                .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = List<State>()


    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.add(state)    }
    }









    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = List<State>()


    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.add(state)    }
    }


    fun assertStates(vararg states: State){


    }


    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = List<State>()


    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.add(state)    }
    }


    fun assertStates(vararg states: State){
         Assert.equals(stateHistory, states.toList())

    }


    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

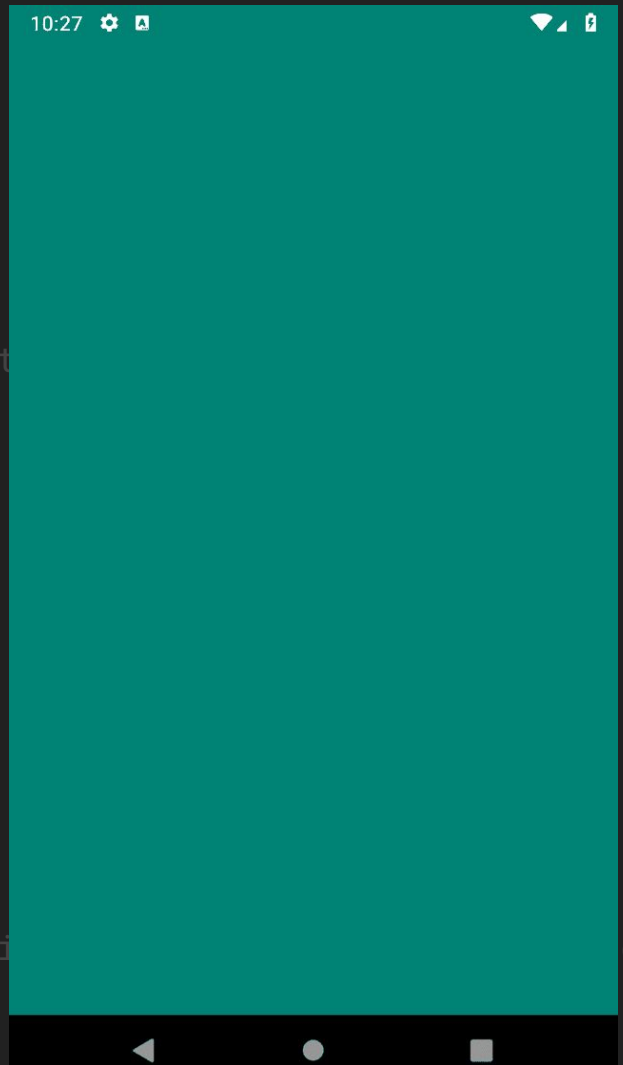
```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertStates(State.Loading)
            assertStates(State.Loading, State.Content(emptyList())

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

         assertStates(State.Loading, State.Content(emptyList()), State.Content(itemToAdd))
        }
    }
}
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertStates(State.Loading)
            assertStates(State.Loading, State.Content(emptyList()))

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

         assertStates(State.Loading, State.Content(emptyList()), State.Content(itemToAdd))
        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertStates(State.Loading)
            assertStates(State.Loading, State.Content(emptyList()))

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertStates(State.Loading, State.Content(emptyList()), State.Content(itemToAdd))
        }
    }
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "A", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertStates(State.Loading)
            assertStates(State.Loading, State.Content(empt

            clickCreateTodoItem()

            createItem {
                enterTitle("A")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertStates(State.Loading, State.Content(emptyLi                d))
        }
    }
```



```
class TodoListRobot {

    private val stateHistory = List<State>()


    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.add(state)    }
    }


    fun assertStates(vararg states: State){
         Assert.equals(stateHistory, states.toList())

    }


    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = List<State>()


    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.add(state)    }
    }









    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = ReplayRelay.create<State>()


    init {
        val viewBinder : ListenableTodoListViewBinder = ...
        viewBinder.stateChangedListener =  { state -> stateHistory.accept(state) }
    }









    fun clickCreateTodoItem() {
        Espresso.onView(ViewMatchers.withId(R.id.newItem))
                .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = ReplayRelay.create<State>()
    private val stateVerifier = StateVerifier(stateHistory)

    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.accept(state) }
    }









    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {









}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {


    @Synchronized
    fun assertNextState(nextExpectedState: S) {









    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState









    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable









    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)









    }
}
```

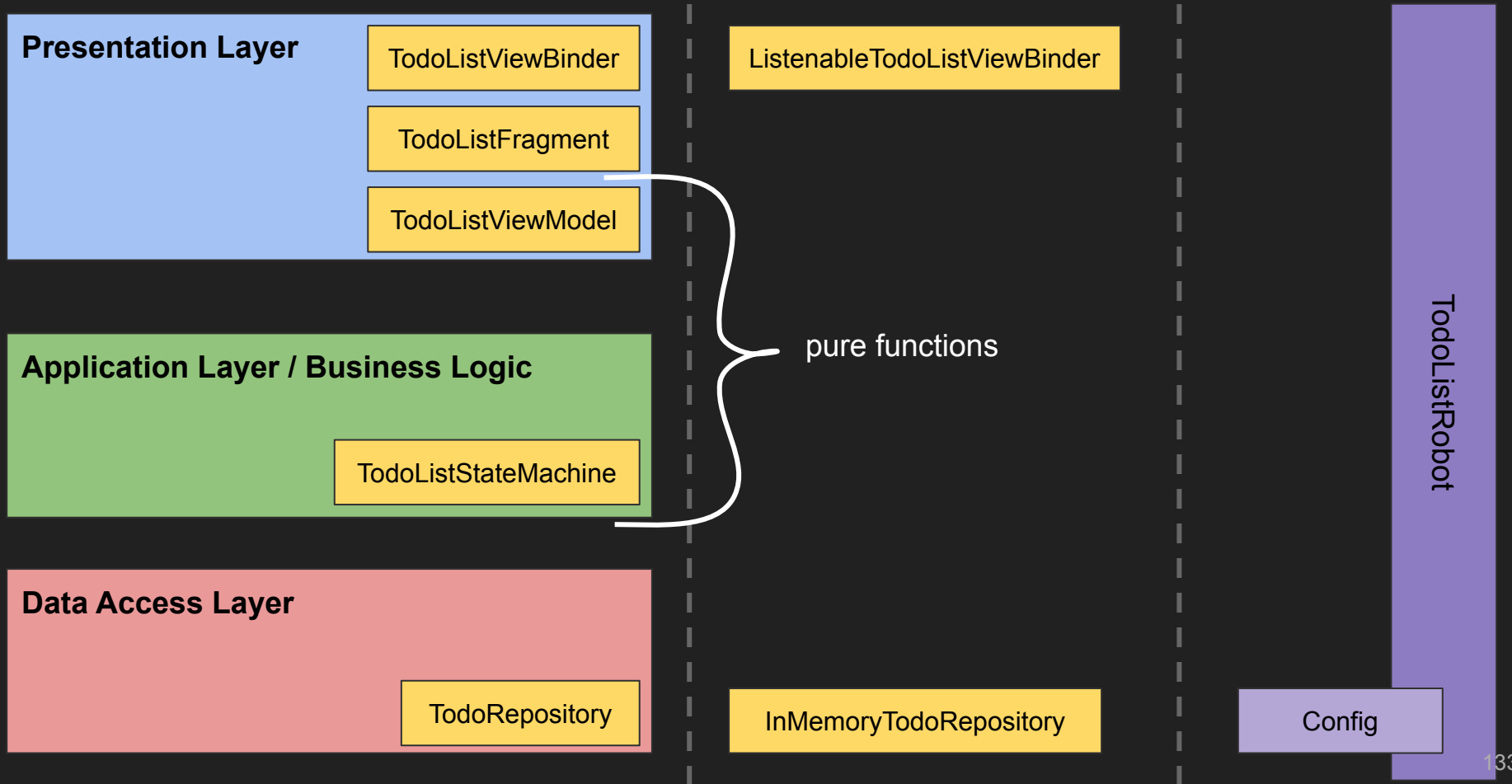
```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)
            .timeout(10, TimeUnit.SECONDS)









    }
}
```

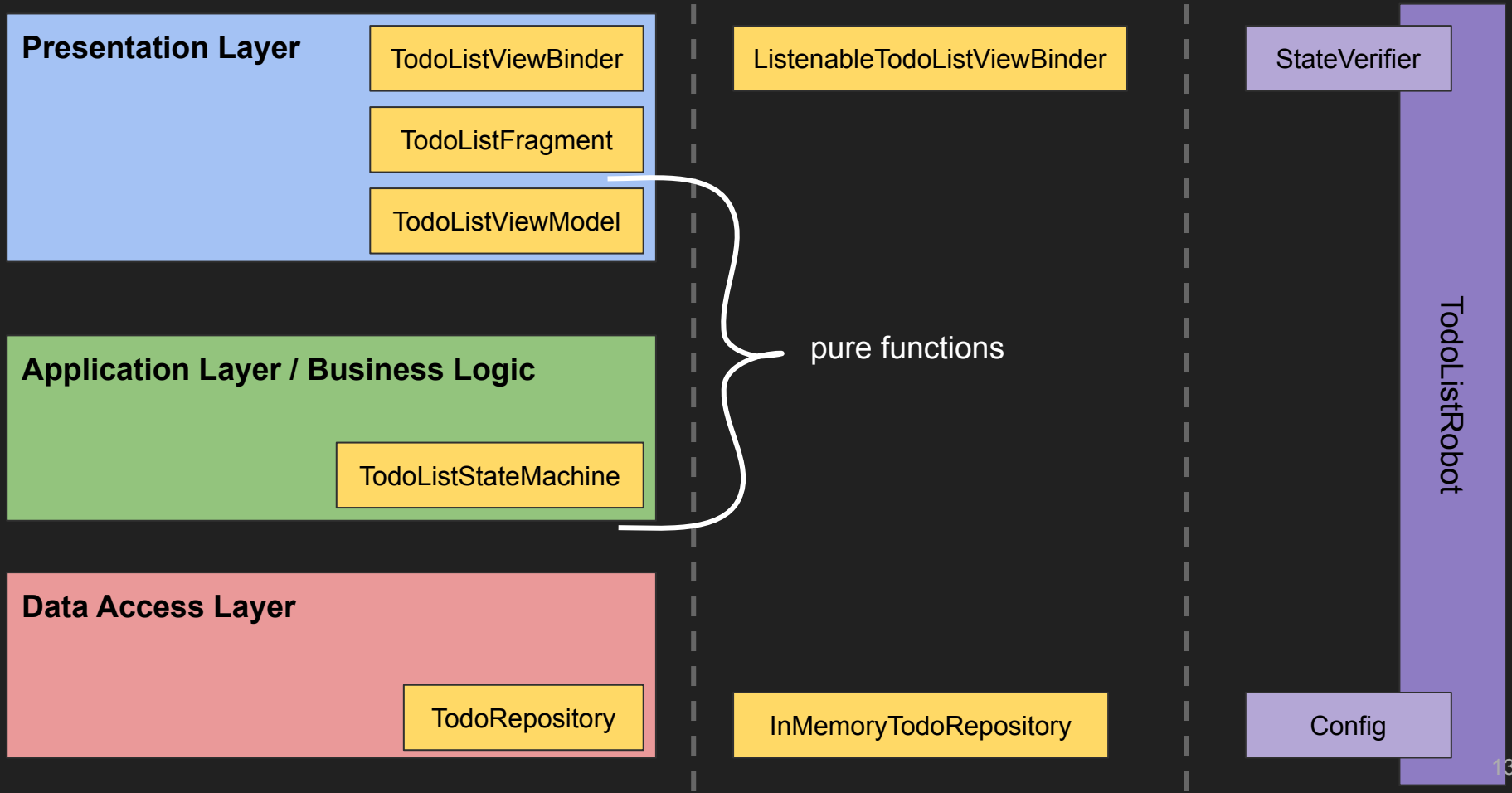
```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)
            .timeout(10, TimeUnit.SECONDS)
            .toList()




    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)
            .timeout(10, TimeUnit.SECONDS)
            .toList()
            .blockingGet()



    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)
            .timeout(10, TimeUnit.SECONDS)
            .toList()
            .blockingGet()

        Assert.assertEquals(expectedStates, actualStates)

    }
}
```

```
private class StateVerifier<S>(private val stateObservable: ReplayRelay<S>) {
    private var alreadyVerifiedStates: List<S> = emptyList()

    @Synchronized
    fun assertNextState(nextExpectedState: S) {

        val expectedStates : List<S> = alreadyVerifiedStates + nextExpectedState
        val actualStates   : List<S> = stateObservable
            .take(alreadyVerifiedStates.size + 1)
            .timeout(10, TimeUnit.SECONDS)
            .toList()
            .blockingGet()

        Assert.assertEquals(expectedStates, actualStates)
        alreadyVerifiedStates = actualStates
    }
}
```


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
ListenableTodoListViewBinder
InMemoryTodoRepository
TodoListRobot
Config


Presentation Layer
TodoListViewBinder
TodoListFragment
TodoListViewModel
Application Layer / Business Logic
TodoListStateMachine
Data Access Layer
TodoRepository
pure functions
ListenableTodoListViewBinder
InMemoryTodoRepository
StateVerifier
Config
TodoListRobot

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "A", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("A")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

```
class TodoListRobot {

    private val stateHistory = ReplayRelay.create<State>()
    private val stateVerifier = StateVerifier(stateHistory)

    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.accept(state) }
    }









    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = ReplayRelay.create<State>()
    private val stateVerifier = StateVerifier(stateHistory)

    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.accept(state) }
    }

    val assertLoadingState: Unit
       get() = stateVerifier.assertNextState(TodoListStateMachine.State.Loading)




    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
class TodoListRobot {

    private val stateHistory = ReplayRelay.create<State>()
    private val stateVerifier = StateVerifier(stateHistory)

    init {
       val viewBinder : ListenableTodoListViewBinder = ...
       viewBinder.stateChangedListener =  { state -> stateHistory.accept(state) }
    }

    val assertLoadingState: Unit
       get() = stateVerifier.assertNextState(TodoListStateMachine.State.Loading)

    val assertContentStateWithEmptyList: Unit
       get() = stateVerifier.assertNextState(TodoListStateMachine.State.Content(emptyList()))


    fun clickCreateTodoItem() {
       Espresso.onView(ViewMatchers.withId(R.id.newItem))
               .perform(ViewActions.click())
    }
}
```

```
@Test
fun newTodoItemIsShownInTodoList() {
    val itemToAdd = TodoItem("1", "Another Item", false)
    given {
        prefilledTodoItems = emptyList()

        todoList {
            assertLoadingState
            assertContentStateWithEmptyList

            clickCreateTodoItem()

            createItem {
                enterTitle("Another Item")
                assertEnterTitleState
                pressNext()
                assertSummaryState
                pressSave()
                assertSavingInProgressState
                assertSavingSuccessfulState
            }

            assertContentState + itemToAdd
        }
    }
```

Observation

No UI verification by using Espresso?

```
class ListenableTodoListViewBinder(private val root: View) : TodoListViewBinder(root) {

    lateinit var stateChangedListener: (State) -> Unit

    override fun render(state: TodoListStateMachine.State) {
        super.render(state)
        stateChangedListener(state)

    }

}
```

```
class ListenableTodoListViewBinder(private val root: View) : TodoListViewBinder(root) {

    lateinit var stateChangedListener: (State) -> Unit

    override fun render(state: TodoListStateMachine.State) {
        super.render(state)
        stateChangedListener(state)
        Screenshot.snap(root).record()
    }

}
```

http://facebook.github.io/screenshot-tests-for-android/

Localization QA

- Freeletics App is localized (9 languages)
- Are all translations correct?
- Use same Robots just without assertions
- Navigate through our App
- Use custom ViewBinder to create screenshots

Q & A

Summary

- Robot Pattern is very readable
- Adding configuration and assertions → reads like specifications
- Test as much production code as possible
- State based Architecture helps
- Robots could be used without real UI (jvm tests)