AutSoft

Android app architecture powered by Jetpack & Kotlin

HWSW mobile! 2018

2018. 11. 22.

Braun Márton Szabolcs
braun.marton@autsoft.hu
@zsmb13

Coroutines

Coroutines

- Language level support for asynchronous, suspendable computations
- Essentially lightweight threads
  - We can start them for almost free
- Allows clients to write code in a traditional, sequential style that they're used to
  - Unlike callbacks, Rx, and other solutions which require an entirely new coding style

```
interface CallbackUserApi {
    @FunctionalInterface
    interface Callback<T> {
        void onResult(T result);
    }

    void getUsers(Callback<List<User>> callback);
}


api.getUsers(new CallbackUserApi.Callback<List<User>>() {
    @Override
    public void onResult(List<User> result) {
        for (User user : result) {
            System.out.println(user.getName());
        }
    }
});
```

```
interface CallbackUserApi {
    @FunctionalInterface
    interface Callback<T> {
        void onResult(T result);
    }

    void getUsers(Callback<List<User>> callback);
}


api.getUsers { users ->
    for (user in users) {
        println(user.name)
    }
}
```

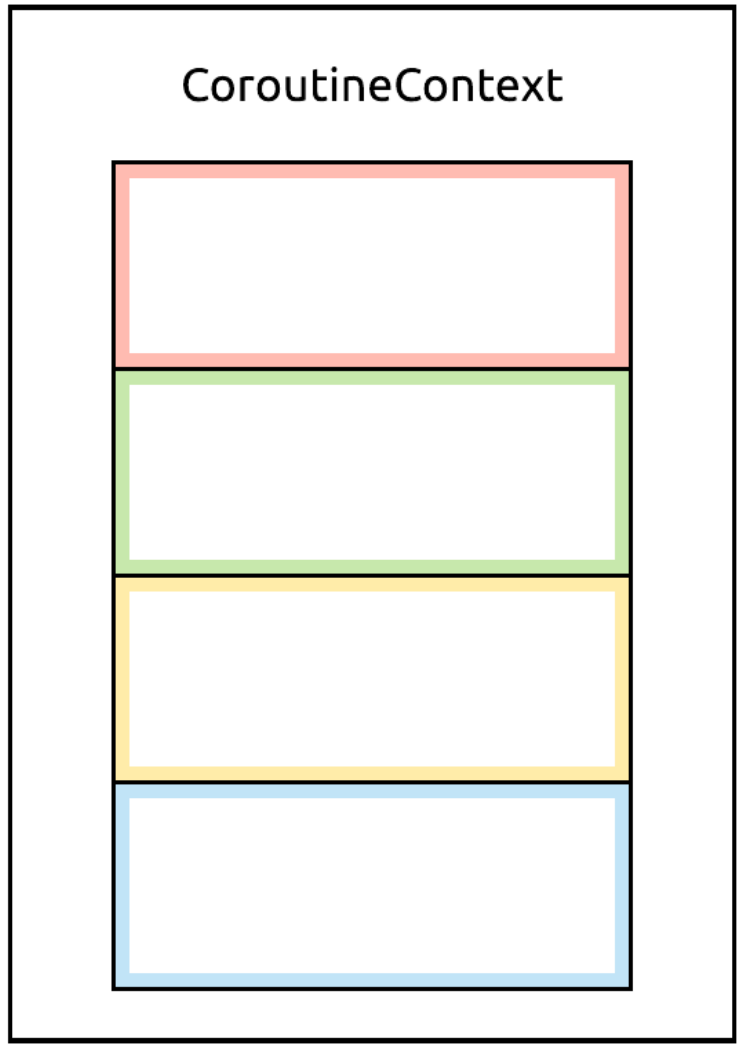
```
interface BlockingUserApi {
    fun getUsers(): List<User>
}
```

```
Thread {
    val users = api.getUsers()
    for (user in users) {
        println(user.name)
    }
}.start()
```

```
launch {
    val users = api.getUsers()
    for (user in users) {
        println(user.name)
    }
}
```

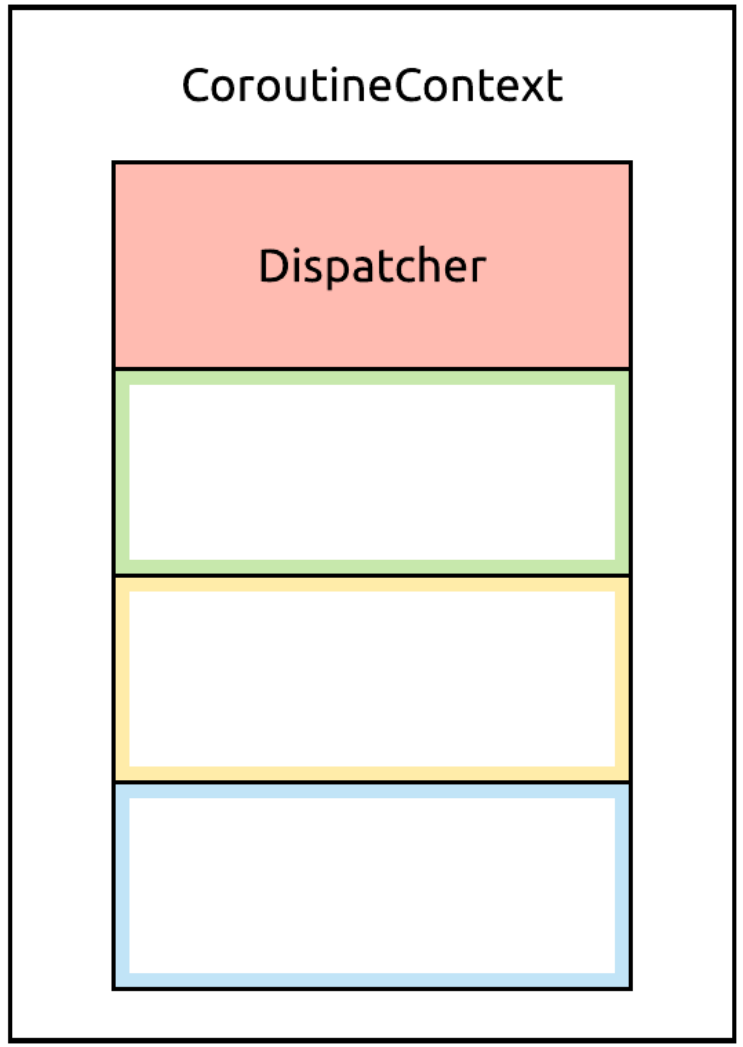
CoroutineContext

- A collection of *elements* that describe *how* the coroutine will be executed
  - Threading
  - Cancellation
  - Error handling
  - Name



Dispatcher

- The most commonly used context element
- Describes what thread the coroutine will run on
  - A single thread, for example the main thread on Android
  - A threadpool containing multiple threads, any of which may be used for the coroutine



Contexts object

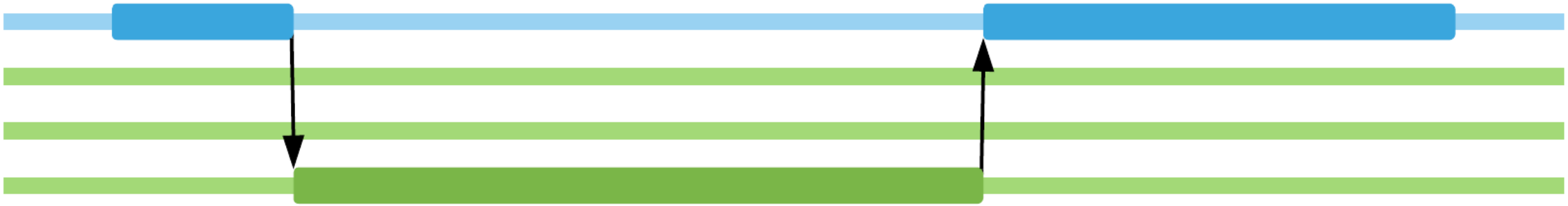
- Our own solution to group the various `CoroutineContexts` and `Dispatchers` in our app
  - Some are built-in `Dispatchers`
  - Some are custom threads and thread pools

```
object Contexts {

    val UI = Dispatchers.Main
    val IO = Executors.newFixedThreadPool(3).asCoroutineDispatcher()

}
```

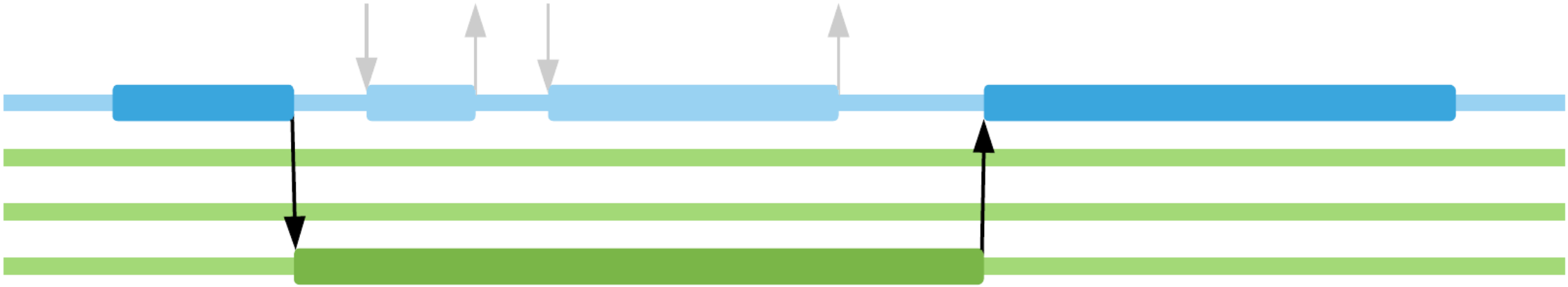
```
launch(Contexts.IO) {
    val users = api.getUsers()
    for (user in users) {
        println(user.name)
    }
}
```

```
launch(Contexts.IO) {
    val users = api.getUsers()

    withContext(Contexts.UI) {
        for (user in users) {
            println(user.name)
        }
    }
}
```

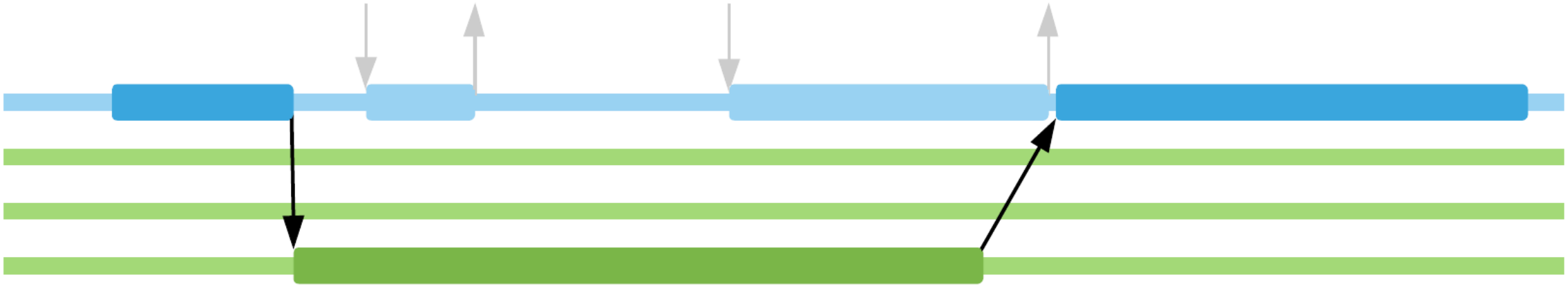
```
launch(Contexts.UI) {
    val users = withContext(Contexts.IO) {
        api.getUsers()
    }

    for (user in users) {
        println(user.name)
    }
}
```



```
launch(Contexts.UI) {
    val users = withContext(Contexts.IO) {
        api.getUsers()
    }

    for (user in users) {
        println(user.name)
    }
}
```

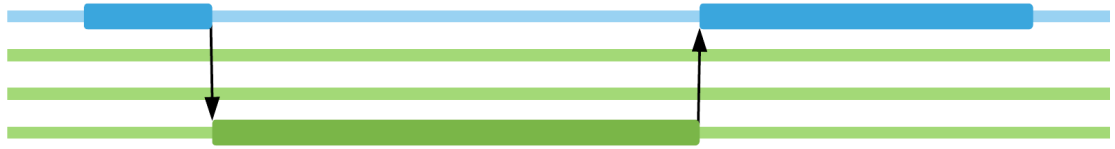


```
launch(Contexts.UI) {
    val users = withContext(Contexts.IO) {
        api.getUsers()
    }

    for (user in users) {
        println(user.name)
    }
}
```



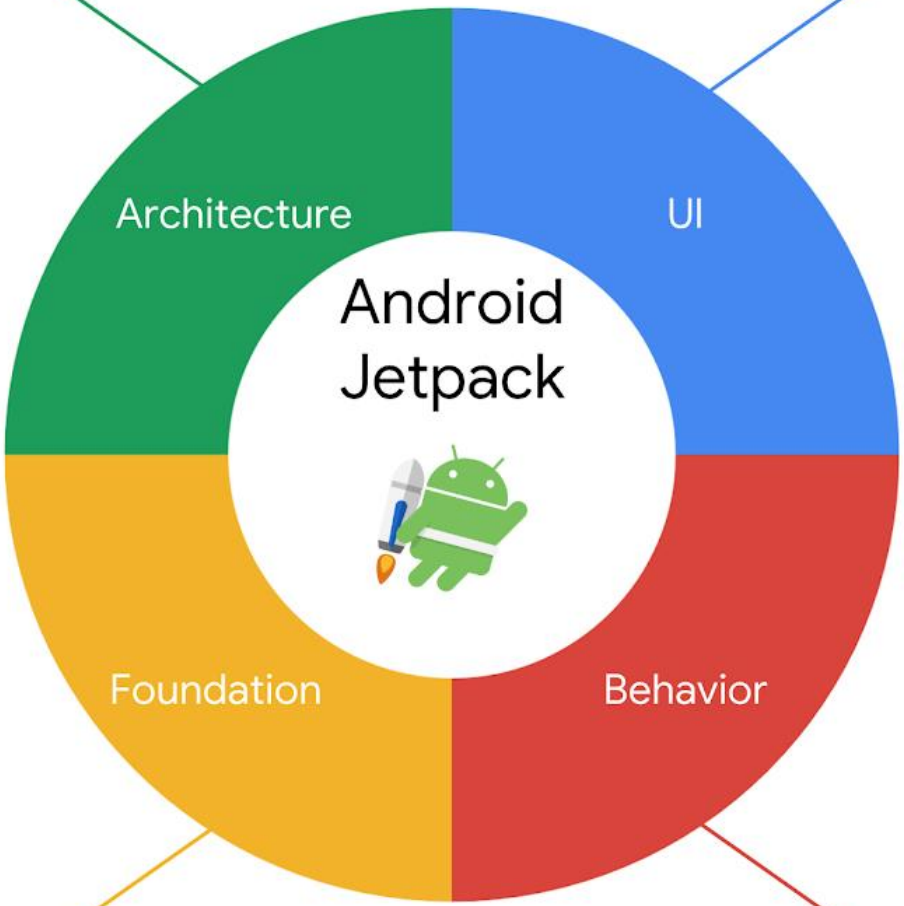
```
class CoroutineUserApi(private val blockingApi: BlockingUserApi) {

    suspend fun getUsers() = withContext(Contexts.IO) {
        blockingApi.getUsers()
    }

}
```

```
launch(Contexts.UI) {
    val users = api.getUsers()
    for (user in users) {
        println(user.name)
    }
}
```



```
class CoroutineUserApi(private val blockingApi: BlockingUserApi) {

    suspend fun getUsers() = withContext(Contexts.IO) {
        blockingApi.getUsers()
    }

}


launch(Contexts.UI) {
    val users = api.getUsers()
    for (user in users) {
        println(user.name)
    }
}
```

Jetpack!



- Data Binding
- Lifecycles
- LiveData
- Navigation *new!*
- Paging *new!*
- Room
- ViewModel
- WorkManager *new!*

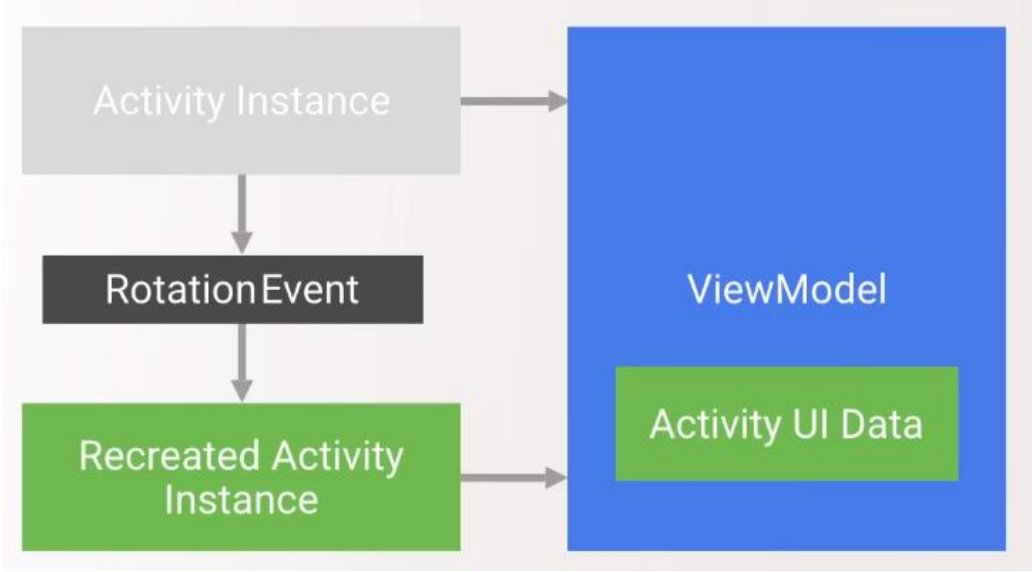
- AppCompat
- Android KTX *new!*
- Multidex
- Test

- Animation & Transitions
- Auto, TV & Wear
- Emoji
- Fragment
- Layout
- Palette

- Download Manager
- Media & Playback
- Permissions
- Notifications
- Sharing
- *new!* Slices

ViewModel

- A component to store UI related state in a lifecycle-conscious manner
  - Should store all UI related state instead of the `Activity`
  - Survives configuration changes, `Activity`/`Fragment` recreation



ViewModel

```
class UserViewModel : ViewModel() {
    val user = User(name ="Sally", age = 25)
}

class UserActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val userViewModel = ViewModelProviders.of(this)
                                    .get(UserViewModel::class.java)
    }
}
```

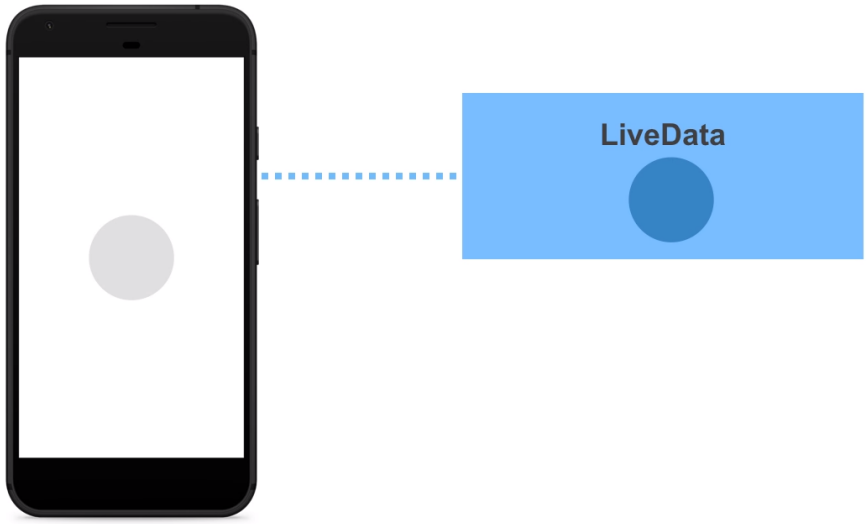
Tips for ViewModel usage

- *Never* hold a reference to a UI element or Activity
  - ViewModels have a longer lifespan, this would be a memory leak
- They don't replace `savedInstanceState` entirely
  - ViewModel: may store any type of data in properties, easy to access from Activities
    - Use it to store all data necessary to populate the UI
  - `savedInstanceState`: survives configuration changes, and even application death on low memory, but stores little data
    - Use it to store the minimal required data to load data into a ViewModel (e.g. an ID)

```kotlin
class UserViewModel : ViewModel(private val context: Context) {
    val user = UserRepository(context).getUser()
}


class UserViewModelFactory(private val context: Context)
     : ViewModelProvider.Factory {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return UserViewModel(context) as T
    }
}


val factory = UserViewModelFactory(applicationContext)
val viewModel = ViewModelProviders.of(this, factory)
                                  .get(UserViewModel::class.java)
```

LiveData

- Lifecycle aware, observable data holder
- Observable:
  - Wraps a single value, and implements the *observer pattern*, notifying any observers when the value it contains changes



LiveData

- Typically used in a ViewModel
  - Observe data in the ViewModel without referencing the Activity

```
class UserViewModel : ViewModel() {
    val user: LiveData<User> = ...
}

class UserActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val userViewModel = ViewModelProviders.of(this).get(UserViewModel::class.java)
        userViewModel.user.observe(this, Observer { user ->
            // show user on UI
        })
    }
}
```

LiveData

- Lifecycle aware
  - Only notifies “active” observers
    - Background Activities don’t get updates
    - When they become active again, they receive only the latest value
  - Observation is automatically cancelled when the `LifecycleOwner`’s lifecycle comes to an end
    - No way to forget, no memory leaks, simple API

```
userViewModel.user.observe(this, Observer { user ->
    // show user on UI
})
```

LiveData

- Instantiation:

```
val user: MutableLiveData<User> = MutableLiveData<User>()
```

- Setting the value from the UI thread:

```
user.setValue(User("Ann", 34))
user.value = User("Jim", 41)
```

- Setting the value from a background thread:

```
user.postValue(User("Zoe", 22))
```

ViewModel & LiveData

- ViewModels usually use backing properties for `LiveData`
  - A private `MutableLiveData` that stores the actual `LiveData` instance holding a value
  - A publicly observable, read-only `LiveData` property that's set to the same instance

```
class UserViewModel : ViewModel() {
    private val _user = MutableLiveData<User>()
    val user: LiveData<User> = _user

    fun initUser() {
        _user.value = User("Michael", 54)
    }
}
```

Cancellation, jobs, parenting, and scope

Cancellation

- Coroutines like we've seen them before will continue running until they throw an exception or complete
- We need a way to cancel them when we no longer need their results
  - `launch` returns a `Job` instance that lets us do this

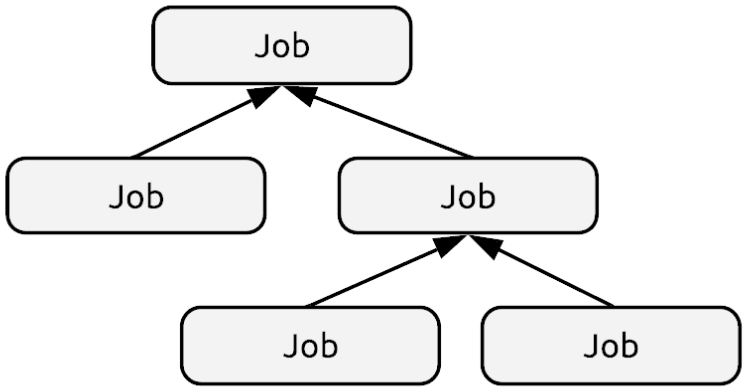
```
val job: Job = launch(Contexts.UI) {
    ...
}

job.cancel()
```

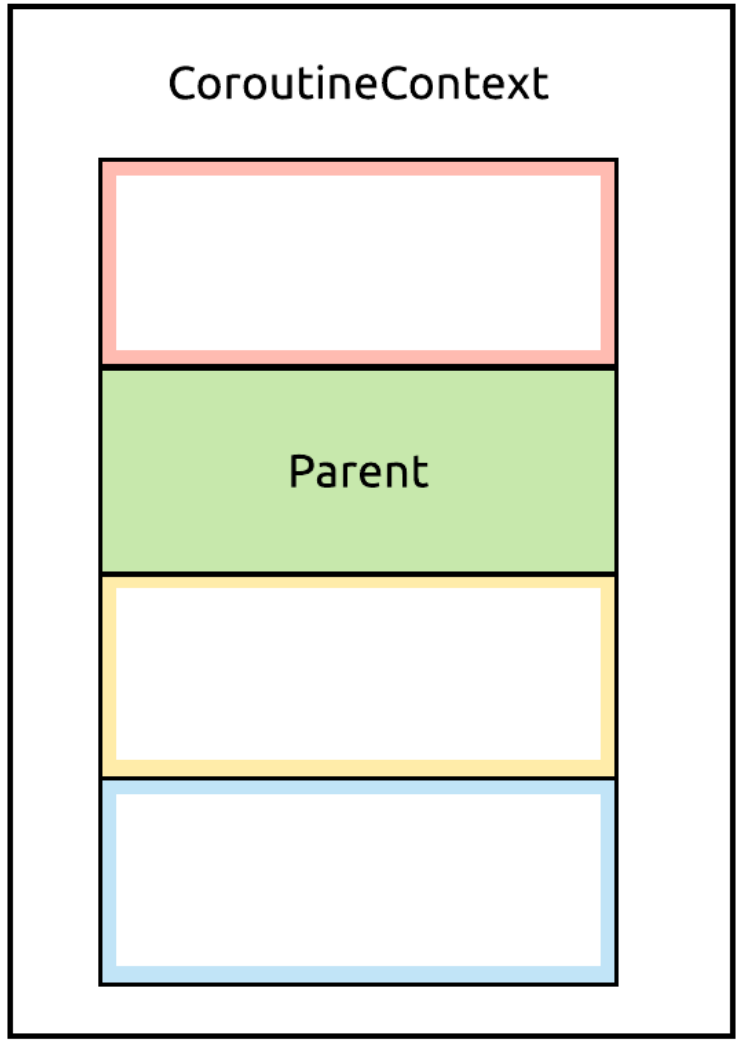
Cancellation

- Where should we launch coroutines?
  - If we launch it in an `Activity`, we'll have to cancel them and start new ones on every rotation, which is wasteful
  - We can start them in our `ViewModels` instead!
    - They'll survive until the screen is active
- How do we manage them?
  - We could put the `Jobs` of every coroutine we launch in a `List`, and cancel all of them when the `ViewModel` is cleared one by one
  - There is, however, a better way

Parent

- The *second* most commonly used context element
- Organizes coroutines in a tree structure
  - If a parent is cancelled, all of its children are cancelled as well



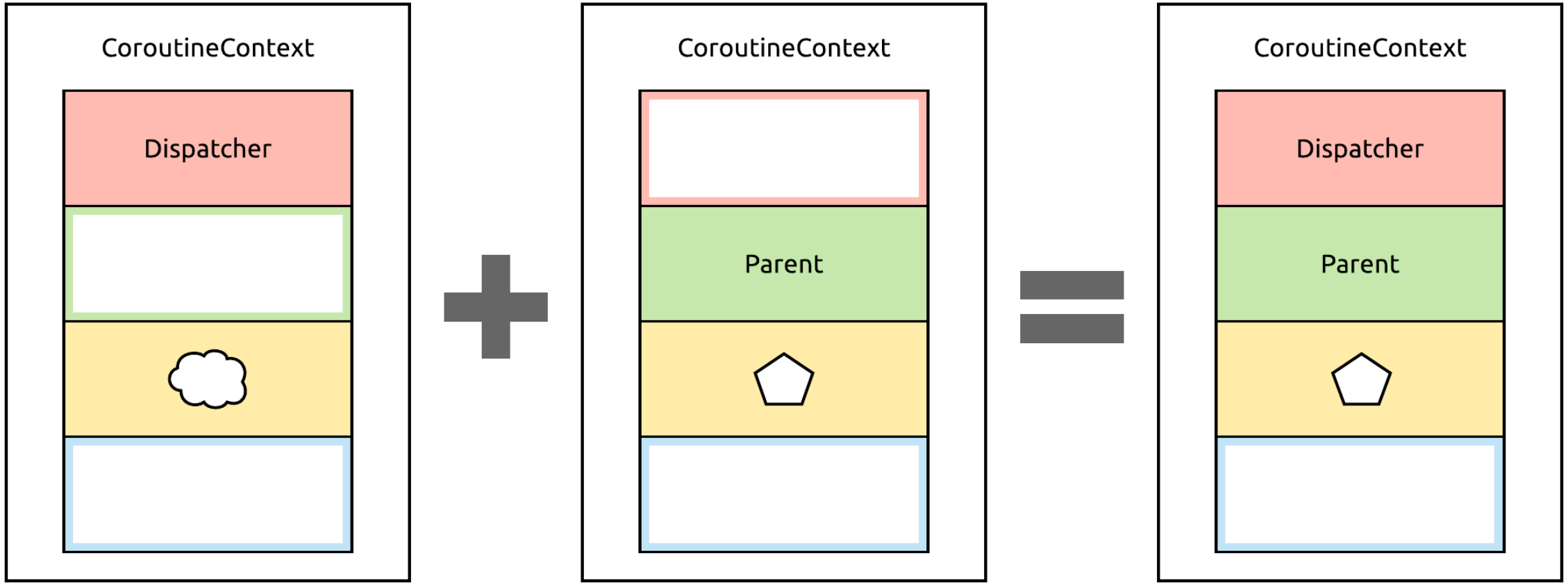


Parent

- A single “empty” parent for all coroutine in a `ViewModel`
  - This lets us cancel them with a single call, in a safe manner

```
class UserViewModel : ViewModel() {

    private val parentJob = Job()

    fun getUserProfile() {
        launch(parentJob) {
            // fetch profile from somewhere
        }
    }

    override fun onCleared() {
        parentJob.cancel()
    }
}
```

Combining CoroutineContexts



Combined contexts

- Using the + operator, we can create a context that defines both the coroutine's parent and `Dispatcher`

```
class UserViewModel : ViewModel() {

    private val parentJob = Job()

    fun getUserProfile() {
        launch(Contexts.UI + parentJob) {
            // fetch profile from somewhere
        }
    }

    override fun onCleared() {
        parentJob.cancel()
    }
}
```

CoroutineScope

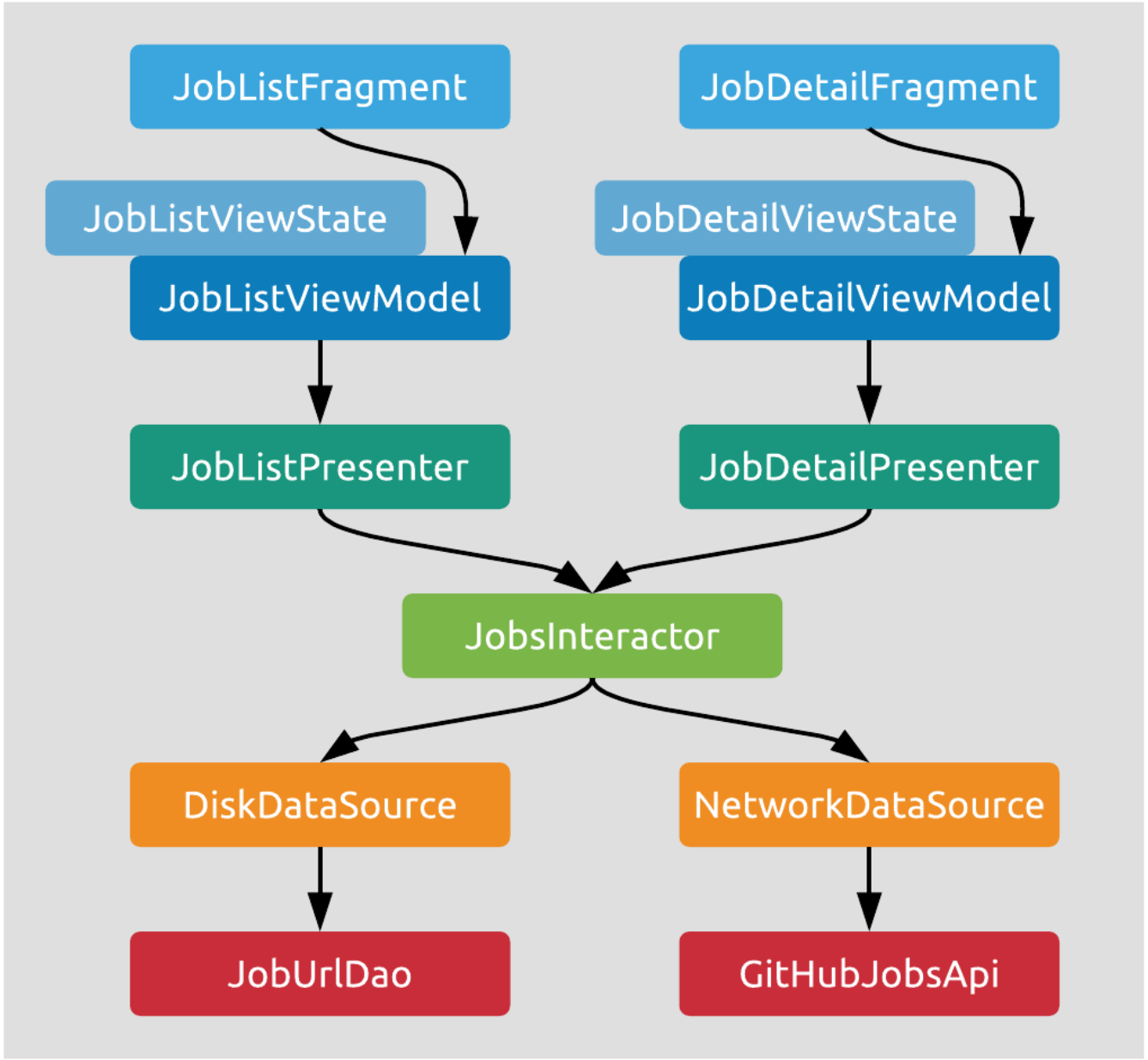
- An interface that defines a common context for all coroutines launched within its scope

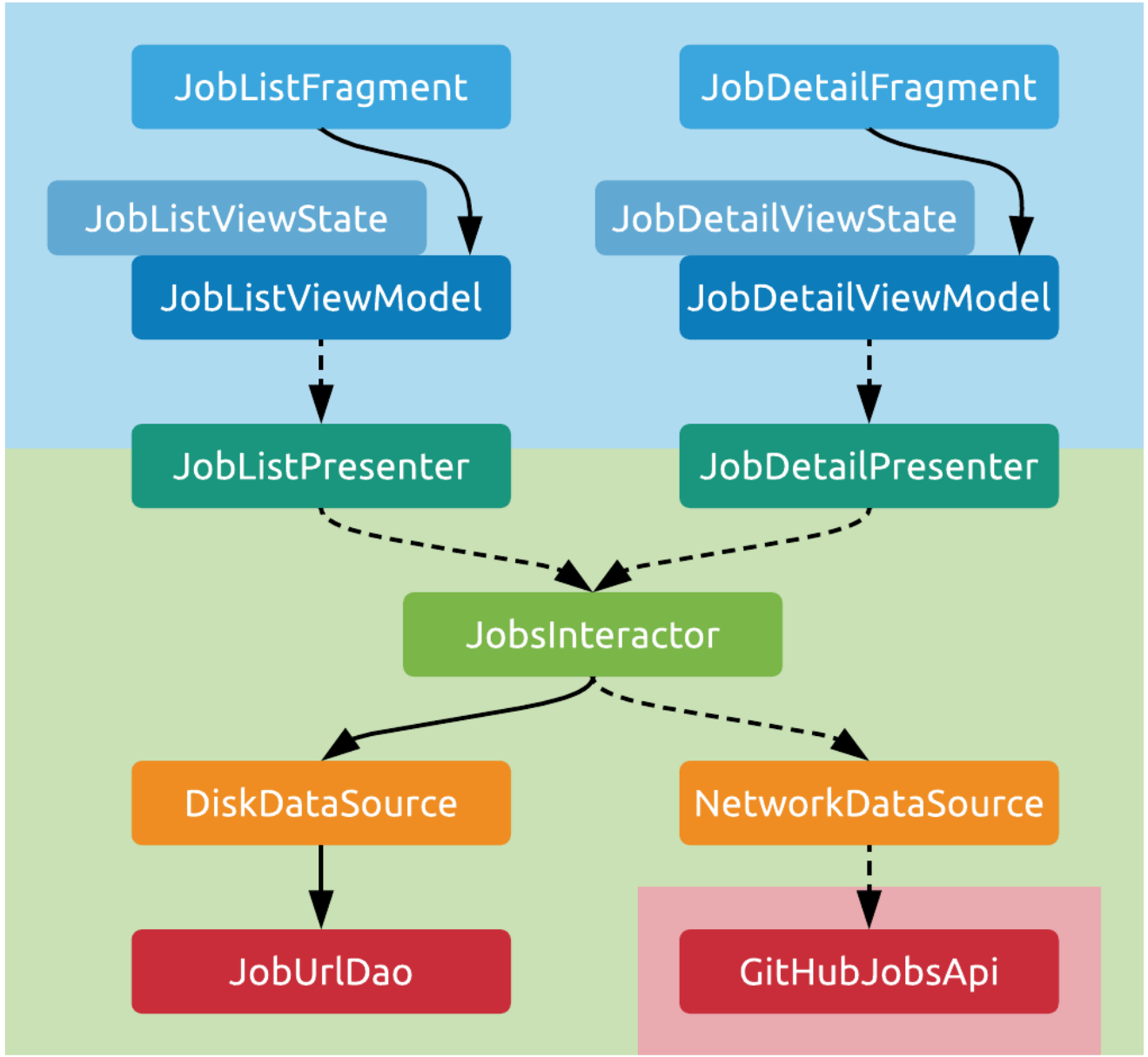
```
class UserViewModel : ViewModel(), CoroutineScope {
    private val parentJob = Job()
    override val coroutineContext: CoroutineContext = Contexts.UI + parentJob

    fun getUserProfile() {
        launch {
            // fetch profile from somewhere
        }
    }

    override fun onCleared() { parentJob.cancel() }
}
```

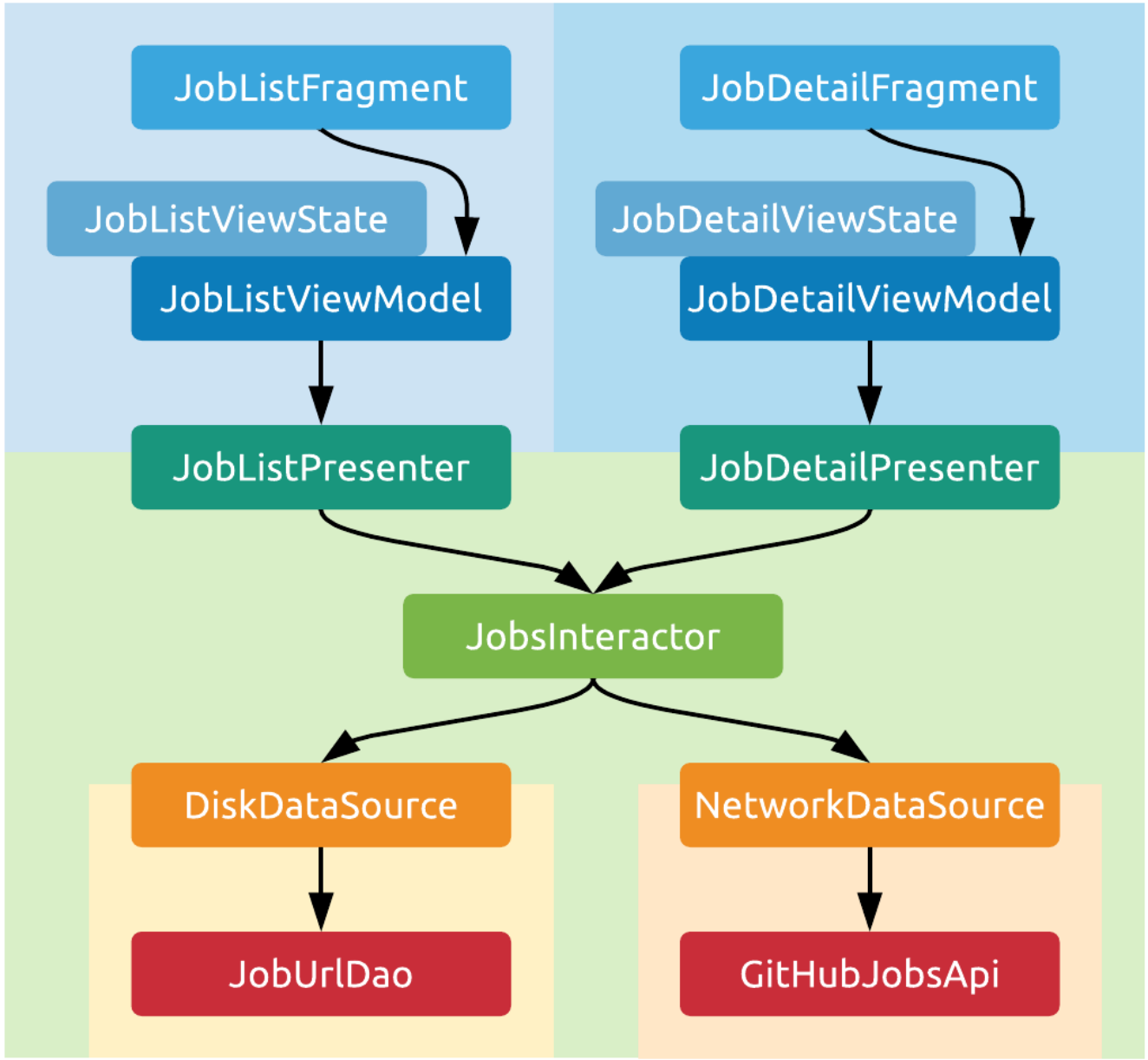
Architecture

Main goals

- Clear separation of responsibilities
  - Well defined coordination of components
- Easy to use threading
  - Offload work to background threads trivially
- Safe state handling
  - Always maintain a consistent state
  - Handle configuration changes and process death gracefully
- Fragment-based navigation in a single Activity
  - With argument passing


JobListFragment
JobDetailFragment
JobListViewState
JobDetailViewState
JobListViewModel
JobDetailViewModel
JobListPresenter
JobDetailPresenter
JobsInteractor
DiskDataSource
NetworkDataSource
JobUrlDao
GitHubJobsApi


JobListFragment
JobDetailFragment
JobListViewState
JobDetailViewState
JobListViewModel
JobDetailViewModel
JobListPresenter
JobDetailPresenter
JobsInteractor
DiskDataSource
NetworkDataSource
JobUrlDao
GitHubJobsApi
Android UI thread
IO thread(pool)
OkHttp threadpool
Blocking calls
Suspending calls



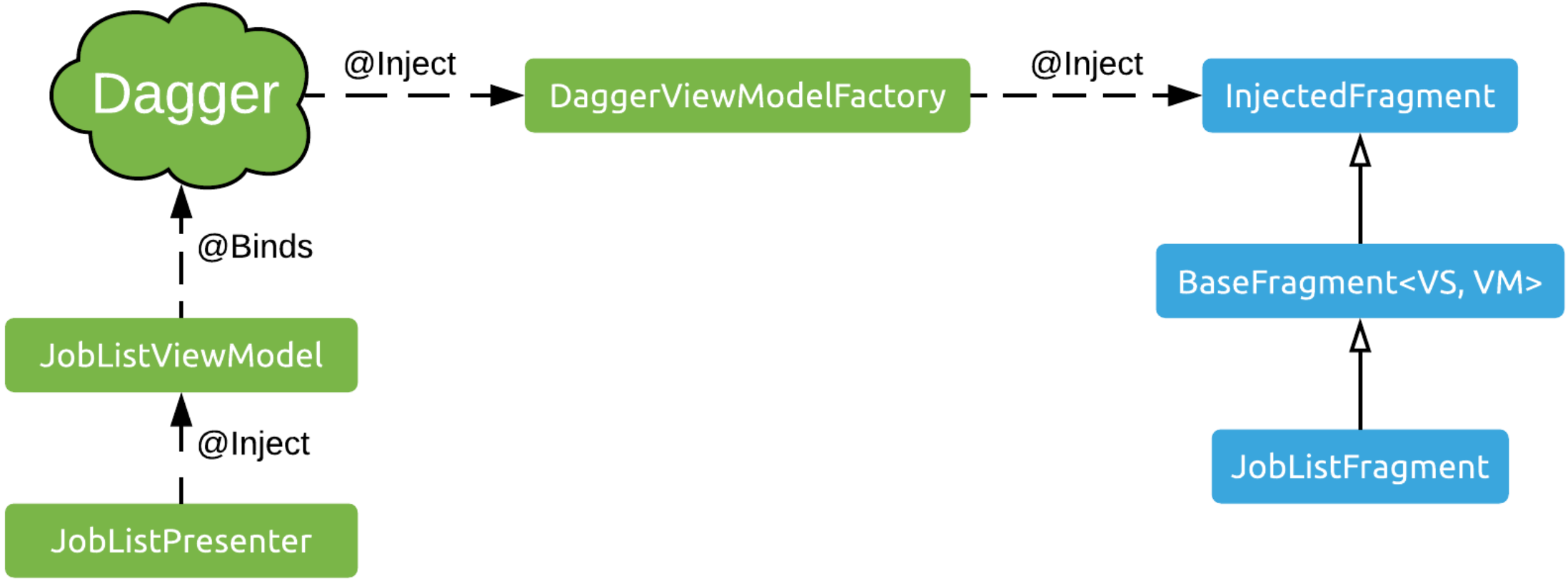
Presentation models

Domain models

DB models

Network models

DaggerViewModelFactory



Thank you for your time!

Android app architecture powered by Jetpack & Kotlin

HWSW mobile! 2018

2018. 11. 22.

Braun Márton Szabolcs

 braun.marton@autsoft.hu  zsmb13