ClockWork Developer Manual

This document was written to teach its readers all of the lessons about Android development that were learned over the first eleven months of ClockWork's development. The intent is to expedite the "norming" period for future developers that are not familiar with the project or Android development. A focus of this document is general tool use and principles so that the requisite concepts are presented directly, instead of needing to uncover them by interpreting applications (although examples are provided in text as well as in the project's source itself).

1. Overview

1.1 Tools Used

1.1.1 Kotlin

The project's code base is practically 100% Kotlin, the only exception being the Gradle scripts.

1.1.1.1 Flows

ClockWork uses a native Kotlin mechanism called Flows to keep all data fetched from the database up to date with the truth. There are two types of Flows: cold Flows (`Flow`) and hot Flows (`StateFlow`); the former will "emit" an update whenever a change occurs while the latter will also keep the latest emitted value cached (`StateFlow.value`). `SharedFlow`s, which enable emissions from multiple sources, are also used to communicate between components asynchronously.

Flow collection means that the flow is perpetually observed for emissions, the values of which are then processed by the defined closure when they occur. Since collection is perpetual, it must be initiated in a coroutine (see below).

1.1.1.2 Delegates

Kotlin has a feature called delegation, which enables using a data structure's `get()` and `set()` functions through a different identifier as if it was a variable. This only makes using these structures more ergonomic; you could use its getters and setters instead without losing any functionality. Delegation is used in ClockWork for lazy dependency instantiation, for accessing the uiState exposed by ViewModels, and for Jetpack Compose's `remember` system (see below).

1.1.1.3 Coroutines

Coroutines are Kotlin's solution for asynchronous programming. They are essentially normal functions that are launched in a `CoroutineScope`, resulting in a `Job` object. Some functions in Kotlin are marked suspend and may only be invoked in a `CoroutineScope`. This leads to `suspend` propagation because any function that

calls a `suspend` function must also be `suspend`. `CoroutineDispatcher`s can be used to create `CoroutineScope`s, but in ClockWork, suspend functions will usually be called by ViewModels (see below), which have their own CoroutineScope (`ViewModel.viewModelScope`) that is linked to their lifecycle and is preferable to using a `CoroutineDispatcher` in most scenarios.

1.1.1.4 The Java API

ClockWork makes use of `java.time.Instant` and `java.time.Duration` in a multitude of places throughout the codebase through Kotlin's interoperability with Java and its runtime. Kotlin does have its own Duration and Instant structures, but they lack a lot of necessary operations and only serve to annoy you when they appear at the top of the autofill suggestion list. Note that some of the `java.time.Instant` and `java.time.Duration` functionality (such as `java.time.Duration.seconds()`) requires a higher API level than the project uses. Upgrading to the requisite level would break compatibility with most devices currently running Android.

1.1.1.5 Data Classes, Data Objects, and Sealed Interfaces

In Kotlin, Data Classes are synonymous with Java 17's Records: immutable structures that are intended to store or transfer data. Like Records, Data Classes come with some special methods, the most notable being `copy()`, which is used to create a new object, using the receiver (Kotlin for the instance the method belongs to) as a template. Using Kotlin's named argument syntax, new values for specific fields can be used with `copy()`.

```
data class Character (
    val name: String,
    val catchPhrase: String,
)

fun main () {
    val steve = Character(
        name = "Steve",
        catchPhrase = "I... am Steve",
    )

    val steve2 = steve.copy(
        catchPhrase = "I am also Steve",
    )
}
```

Data Objects are unit structures; they have no fields. As such, each data object definition is (excluding reflection) as singleton, a single instance is reused across all appearances. There exists only one of each defined data object.

By combining data classes and data objects with Sealed Interfaces, tagged union types can be defined. Tagged unions are powerful tools to express irregular or asymmetrical state domains without introducing the possibility of invalid states. Consider the following fields to manage the state of the timer:

```
private enum class State {
    DORMANT, PREPARING, WORKING, ON_BREAK, CLOSING
}

private var internalState = State.DORMANT

private var loadedTask: StartedTask? = null
private var loadedTaskId: Long? = null

private var elapsedWorkSeconds = 0
private var elapsedBreakSeconds = 0
```

As indicated by the State enum, the timer can be in any of five high-level states at a time. Each state variant has different responsibilities which require the other fields to manage. For instance, the PREPARING state needs the loadedTaskId while the Task is asynchronously fetched. RUNNING and PAUSED need the loadedTask field as well as the elapsed time fields to track the incrementations. The values of the backing fields must always be consistent with the value of internalState, otherwise the timer has entered into an invalid state and might trigger a failure (consider `loadedTask = null` while `internalState = State.RUNNING`). Additionally, even though the values of some fields do not matter if the current state does not use them, the presence of irrelevant fields widens the representational gap, which makes the program more vulnerable to logical errors. Consider this alternative:

```
private sealed interface InternalState {
   data object Dormant : InternalState
   data object Closing : InternalState
   data class Preparing(
       val taskId: Long,
   ) : InternalState
   data class Working(
       val task: StartedTask,
       val elapsedWorkSeconds: Int,
       val elapsedBreakSeconds: Int,
   ) : InternalState
   data class OnBreak(
       val task: StartedTask,
       val elapsedWorkSeconds: Int,
       val elapsedBreakSeconds: Int,
```

```
    ) : InternalState
}

private val internalState = InternalState.Dormant
```

With this model, it is impossible to represent a state that is nonsensical or invalid. Backing fields are only present when relevant as well, shrinking the gap. One more benefit with this model is that state transitions are atomic; state changes in the previous model required a sequence of transitions between intermediate, potentially invalid, states while each field was updated individually.

1.1.2 Jetpack Compose & Material 3

Jetpack Compose is the declarative GUI library ClockWork uses. GUI elements are expressed as Kotlin functions with the `@Composable` annotation. Material 3 GUI components, which are integrated into Compose, are used throughout the project.

When using Android Studio, previews of Composable functions can be rendered without launching the app by using the `@Preview` annotation. Throughout ClockWork, dedicated, private preview functions are placed alongside the corresponding Composable so that it can easily be tested.

Composable functions are rendered into graphical interfaces in a process called Composition, or recomposition. Composition essentially discards the existing render to render the composable anew, and all local variable values are initialized. This loss of state can be circumvented using Compose's `remember` system. Instead of initializing a variable using "`= rvalue`", use "`by remember { rvalue }`". Some pre-made widget state holders have their own remember functions you should use instead of instantiating the state holder in `remember {}` directly e.g. `rememberScrollState()`.

1.1.3 Room

Android Room is a SQLite Kotlin wrapper. Tables are defined as data classes and interfaces (called Data Access Objects or `DAO`s) are used to define functions that perform database CRUD operations.

1.1.3.1 Entities

Entities are database tables. Use the `@Entity` annotation and a data class to define the fields for the table. The name of the class will be used for the table name in SQL queries. For best compatibility, restrict the field types of the data class to those native to or trivially coercible to SQLite databases (e.g. `Long`, `Int`, `String`, `Boolean`), otherwise functions marked `@TypeConverter` that convert from the complex data type to the SQLite primitive (and vise versa) must be defined in the class provided to the database using the `@TypeConverters` annotation.

The `@PrimaryKey` annotation is used to mark a field as a primary key and, if the field is an `Int` or `Long`, the annotation's `autoGenerate` argument can be set so that it automatically increments when inserting new records, so long as the corresponding field in the record is set to 0. The `@Entity` annotation arguments can be used to define foreign keys and indices for the table:

```
@Entity(
    foreignKeys = [ForeignKey(
        entity = ProfileEntity::class,
        parentColumns = ["id"],
        childColumns = ["profileId"],
        onDelete = ForeignKey.SET_NULL
    )],
    indices = [Index(value = ["profileId"])]
)
data class TaskEntity(
    @PrimaryKey(autoGenerate = true) val taskId: Long = 0,
    // The rest of the fields...
)
```

1.1.3.2 Daos

Data Access Objects (DAOs) are interfaces annotated `@Dao` where the database CRUD operations are declared, but not necessarily defined. To declare a DAO query to perform a CRUD operation, write a corresponding Kotlin function signature with the arguments (usually an entity field or an entire entity) needed to perform the operation and the return type. The return type can be omitted if one is not desired (e.g. insert functions return the rowid). Use CRUD annotations such as `@Insert`, `@Update`, or `@Query` to mark the signature with the correct behavior. `@Insert`, `@Update` are "convenience annotations" that generate the query at compilation by interpreting the parameters and return types. `@Query` is used to write SQL directly, and function arguments can be accessed with `:paramName`. Use `@Transaction` in addition to the CRUD annotation to make the function atomic. The return type of the DAO function needs to be consistent with the results of select queries else the operation will fail. Select queries that may find multiple records must have `List` return types, and queries for individual records must have nullable return types for when the query does not find a matching record.

DAO queries that perform an action and return once (e.g. inserting, updating, deleting, queries that return records directly) are called "one-shot" and must be marked `suspend`. Their counterpart, "observable queries," return a living connection to records

in the database that reflect their values as they change. These use some sort of observation mechanism, `Flow`s in ClockWork, and do not need to be asynchronous.

```
@Dao
interface TaskDao {
   @Insert(onConflict = OnConflictStrategy.ABORT)
   suspend fun insertTask(taskEntity: TaskEntity): Long

   @Update
   suspend fun updateTask(task: TaskEntity)

   @Transaction
   @Query("DELETE FROM TaskEntity WHERE taskId = :id")
   suspend fun deleteTask(id: Long)

   // And so forth...
}
```

To use a DAO, create a public abstract function in the database class (that extends RoomDatabase) whose return type is the DAO.

1.1.3.3 Embedded Objects

Embedded objects are a mechanism to express a type as a composition of database entities. For instance, in ClockWork, a task session has properties that describe the task (name, estimate, etc.) as well as a list of segments that describe execution and markers that occurred over its execution. In the database, these are stored in three separate tables because of the two many-to-one relationships. These relationships can be captured using embedded objects to create a data class that Room can use to automatically assemble a complete task session from the separate records.

```
data class TaskWithExecutionDataObject(
   @Embedded val taskEntity: TaskEntity,

   @Relation(
       parentColumn = "taskId",
       entityColumn = "taskId"
   )
   val segments: List<SegmentEntity>,

   @Relation(
       parentColumn = "taskId",
       entityColumn = "taskId"
```

```
    )
    val markers: List<MarkerEntity>,
)
```

When used as a return type in a DAO function, Room will perform the necessary joins using the relation defined in the embedded object's fields to construct the object.

```
@Transaction
@Query("SELECT * FROM TaskEntity WHERE taskId = :taskId")
fun getTask(taskId: Long): Flow<TaskWithExecutionData?>
```

1.1.3.4 Migrations

On compilation, Room will build the database schema based on the registered entities. Room requires that distinct schemas use different version numbers and will refuse to finish compiling if it detects a change to the schema without an increment to the version. When the version is incremented properly, Room will apply a migration to adapt the data in the existing schema to the new version if one is defined.

Room migrations are defined by extending the Migration class, specifying the to and from versions, then overriding the migrate() function. The migration logic itself is SQL DDL and DML that apply the changes in the schema in addition to whatever is needed to preserve the data.

1.1.4 Apache Commons Math 3

The Linear Regression implementation from `org.apache.commons:commons-math3` used by the `edit_session_feature` to calculate averages based on session parameters.

1.1.5 Compose Charts

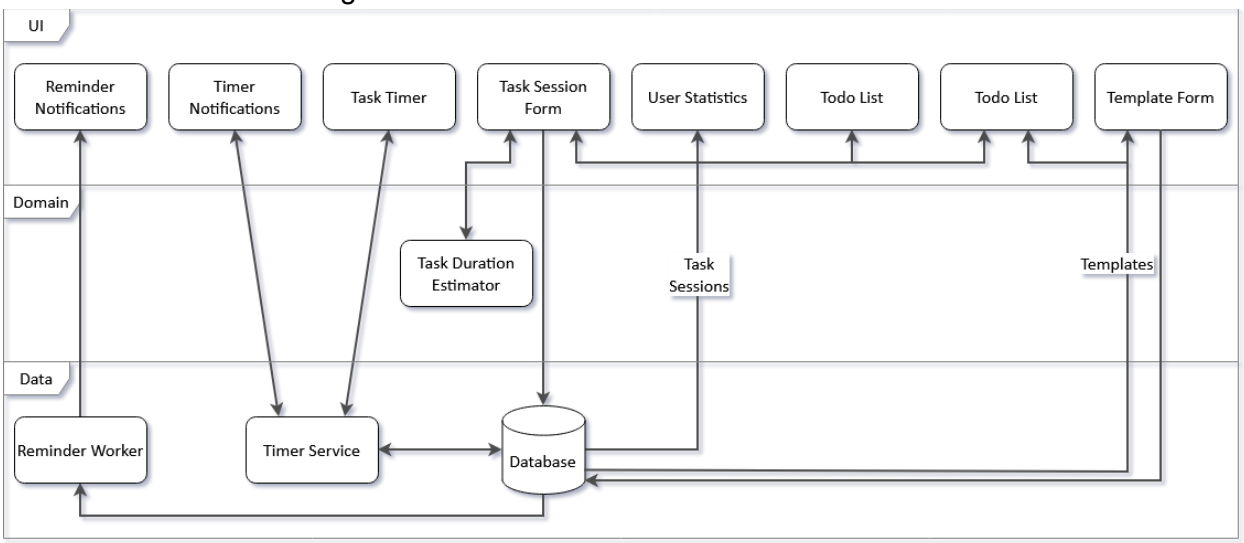
The Line chart implementation from `io.github.ehsannarmani:compose-charts` used by the `user_stats_feature` to show user estimate accuracy for completed sessions.

1.1.6 JUnit

Used to run unit and integration tests. A test in JUnit is a function annotated `@Test`. In the body of the test function, processes are executed and their results tested against the expected results using assert functions (e.g. `assertTrue()`, `assertEquals()`). Tests that can be executed independently from Android are called Unit tests and are placed in the "test" directory under "src." Their counterpart, Instrumented tests are placed in the "androidTest" directory under "src." Both types of tests are driven by JUnit, using JUnit's annotations and assert functions.

1.2 Architecture & Organization

The following diagram depicts a simplified representation of ClockWork's dataflow architecture at a high level.



1.2.1 MVVM

Model-View-Viewmodel: View and Model are separated by a component called the ViewModel which acts as a translation/abstraction layer. The ViewModel transforms data from the Model for the View to present and handles events from the View and delegates them to the Model while keeping both sides independent from each other. The ViewModel also makes the state of the UI more resilient. For instance, data handled by the ViewModel is not lost on screen rotations.

1.2.2 Clean Architecture

Hierarchical architecture structure where dependencies can only point inward (the Dependency Rule). The Domain layer is the innermost and depends on nothing. The Data layer is above the Domain and depends on it. In some interpretations, the Presentation/UI layer is above the data layer and depends on it and the Domain, but in ClockWork, the UI only depends on the Domain layer.

1.2.3 Unidirectional Data Flow

The UI and its state holder (its ViewModel) communicate using state and event data structures: the state holder emits state updates and the UI calls the state holder's `onEvent(event)` function and passes an event. This pattern was adopted late into the project's development and some components do not implement it. Some state holders also use effect data structures to signal reactions in the UI.

1.2.3.1 UI State Construction, Exposure, and Collection

Every ClockWork page ViewModel implements the same pattern for uiState construction, exposure, and collection.

The uiState is exposed from the ViewModel through an immutable StateFlow, which is supplied by a MutableStateFlow which is updated as changes to the state are made. Each page's uiState has a "Retrieving" variant to represent the ViewModel waiting for data to be fetched from the database. This is the initial value of the StateFlows.

```
private val _uiState = MutableStateFlow(TimerUiState.Retrieving)
val uiState: StateFlow<TimerUiState> = _uiState.asStateFlow()
```

The uiState construction may differ depending on the nature of data the page requires. If the page requires living data from a data source (e.g. the timer or the database) then all observers are combined into a single Flow. The Flow emissions are transformed into proper state based on their values. This process is specific to each ViewModel and uiState. Finally, the private uiState MutableStateFlow is updated with the new uiState. This process is performed in a coroutine launched from the viewModelScope when the ViewModel first loads so that it can run asynchronously and continuously so long as the page is active.

```
init {
    viewModelScope.launch {
        combine(
            loadedTask,
            timerState
        ) { task, timerState ->

            if (task == null || timerState == null) {
                return@combine TimerUiState.Retrieving
            }
        // The rest of the transformation logic that produces
        //  a uiState on all paths...
        }
        .collect { newState ->
            _uiState.update { newState }
        }
    }
}
```

Pages that depend on the ViewModel can access the uiState it exposes through the uiState StateFlow, but every page in ClockWork also transforms it into a State so that Compose is aware of updates and delegated to be more ergonomic to use. The resulting uiState is then passed to the subsequent Composables that need it to build the GUI (see State Hoisting).

```
val uiState by timerViewModel.uiState.collectAsStateWithLifecycle()
```

1.2.3.2 Effect Emission, Exposure, and Response

While most pages in ClockWork do not use effects, those that do follow the same pattern.

Effects are exposed symmetrically to uiState: with a private MutableSharedFlow that backs a public immutable SharedFlow.

```
private val _effect = MutableSharedFlow<SessionFormEffect>()
val effect = _effect.asSharedFlow()
```

Instead of a single centralized asynchronous block that manages the uiState independent from other ViewModel business, effects are managed in-place by launching a short-lived coroutine using the viewModelScope that emits an effect variant through the MutableSharedFlow where appropriate. For instance, the following is an excerpt from the onSaveClick() function from the task session form:

```
// ...
if (taskName.isBlank()) {
    viewModelScope.launch {
        _effect.emit(SessionFormEffect.ShowSnackbar(
            "Failed to save session: Missing Name"
        ))
    }
    return
}
// ...
```

Contrasted with uiState, ClockWork pages are bare more immediate responsibility for handling effects. The top-level page composables collect the exposed effects flow, much like the uiState, but instead of passing down the composables chain, emitted effect values are immediately handled through type matching. Collection requires a thread launched from a CoroutineScope, which is provided using a LaunchedEffect Composable. Like the flow combination from uiState construction, this process is continuous until the page is deactivated.

```
LaunchedEffect(Unit) {
    viewModel.effect.collect { effect ->
        when (effect) {
            is SessionFormEffect.ShowSnackbar -> {
                snackbarHostState.showSnackbar(effect.message)
            }
```

```
            is SessionFormEffect.NavigateBack -> {
                onBackClick()
            }
        }
    }
}
```

1.2.4 Manual Dependency Injection

Collaborating components are passed into the dependent component (e.g. a UI page is passed its ViewModel; the page depends on the ViewModel but the ViewModel does not depend on the page). It is standard practice to automate this process with libraries like Hilt, but Clockwork does it manually. This means that all collaborators are deliberately instantiated and managed and then passed into the dependent components on their instantiation. See the `AppContainer` in the `MainApplication`.

1.2.4.1 AppContainer

The `AppContainer` is a class that holds and manages all of the dependencies used throughout the app this currently includes:

- `TimerRepository`
- `RestoreTimerObserver`
- `TimerNotificationManager`
- `ReminderNotificationManager`
- `SessionReminderScheduler`
- `PermissionRequestSignaller`
- `AppPreferencesRepository`
- `GetSessionUseCase`
- `GetAllSessionsUseCase`
- `GetAllCompletedSessionsUseCase`
- `GetAllTodoSessionsUseCase`
- `GetAllSessionsForProfileUseCase`
- `GetActiveSessionIdUseCase`
- `AddMarkerUseCase`
- `EndLastSegmentAndStartNewUseCase`
- `StartNewSessionUseCase`
- `CompleteStartedSessionUseCase`
- `CreateSessionUseCase`
- `UpdateSessionUseCase`
- `DeleteSessionUseCase`
- `GetProfileUseCase`
- `GetAllProfilesUseCase`

- `CreateProfileUseCase`
- `UpdateProfileUseCase`
- `DeleteProfileUseCase`
- `GetRemindersForSessionUseCase`
- `ProcessScheduledReminderUseCase`
- `CalculateEstimateAccuracyUseCase`
- `GetAverageEstimateErrorUseCase`
- `GetAverageSessionDurationUseCase`
- `ManageFirstLaunchUseCase`

Since `AppContainer` is responsible for initializing the dependencies, it also manages their dependencies:

- The "`applicationScope`" which is just a `CoroutineScope` that all system dependencies use
- `TaskRepository`
- `ProfileRepository`
- `ReminderRepository`
- `TimerServiceIntentFactory`
- `TimerNotificationActionProvider`
- `SharedPreferences`
- `GetAppEstimateUseCase`

While not formally designed as such, these objects are practically singletons that are shared between the dependent components of the system.

1.2.4.2 ViewModel Instantiation

As written above, ViewModels are dependencies of the app's user interface that are responsible for managing collaboration between the UI layer and the domain and/or data layer (masquerading as the domain layer). The lifecycle of ViewModels is tied to that of the UI, and as such (and for other reasons such as capturing navigation arguments) they need to be especially instantiated.

All ViewModels in ClockWork define their own factory using the following pattern. The initializer uses a `CreationExtras` scope which contains a key to access the `MainApplication` class from its entry map. From there, the app dependencies can be obtained.

```
class AppViewModel(
    private val dependency1: Type,
    private val dependency2: Type,
): ViewModel() {
    // ViewModel business

    companion object {
        val Factory: ViewModelProvider.Factory = viewModelFactory {
```

```
            initializer {
                val appContainer = (this[APPLICATION_KEY as
                    MainApplication).appContainer

                AppViewModel(
                    dependency1 = appContainer.dependency1,
                    dependency2 = appContainer.dependency2,
                )
            }
        }
    }
}
```

A page's ViewModel is then instantiated in its `NavGraph` declaration as follows. The `AppPageRoute` is a `@Serializable` data object or class that represents the page in the `NavGraph`.

```
fun NavGraphBuilder.appPage() {
   composable<AppPageRoute> { backStackEntry ->

       val appViewModel = ViewModelProvider.create(
           store = backStackEntry.viewModelStore,
           factory = AppViewModel.Factory,
           extras = backStackEntry.defaultViewModelCreationExtras
       )[AppViewModel::class]

       AppPage(
           viewModel = appViewModel,
       )
   }
}
```

If the ViewModel has parameters that are captured from navigation (such as the id of a task), then a key must be defined statically in the ViewModel so that it can be associated with the value we want in the `CreationExtras` map:

```
class AppViewModel(
    private val taskId: Long,
    private val dependency1: Type,
    private val dependency2: Type,
): ViewModel() {
    // ViewModel business
```

```
    companion object {

        val TASK_ID_KEY = object : CreationExtras.Key<Long> {}

        val Factory: ViewModelProvider.Factory = viewModelFactory {
            initializer {
                val appContainer = (this[APPLICATION_KEY as
                    MainApplication).appContainer

                val taskId = this[TASK_ID_KEY] as Long

                AppViewModel(
                    taskId = taskId,
                    dependency1 = appContainer.dependency1,
                    dependency2 = appContainer.dependency2,
                )
            }
        }
    }
}
```

And the corresponding ViewModel instantiation is as follows. In this case, the route is a data class marked `@Serializable` with a `Long` field named `id`. The id provided in an instance of the route class that is given as a parameter in a call to `NavController.nagivate()`. The value of `id` is then associated with the key from the ViewModel in the provided `CreationExtras`. The `defaultViewModelCreationExtras` from the `NavBackStackEntry` is needed to get a reference to the `MainApplication`.

```
fun NavGraphBuilder.appPage() {
   composable<AppPageRoute> { backStackEntry ->

        val taskId = backStackEntry.toRoute<AppPageRoute>().id

        val appViewModel = ViewModelProvider.create(
            store = backStackEntry.viewModelStore,
            factory = AppViewModel.Factory,
            extras = MutableCreationExtras(
                backStackEntry.defaultViewModelCreationExtras
            ).apply {
```

```
                set(AppViewModel.TASK_ID_KEY, taskId)
            }
        )[AppViewModel::class]

        AppPage(
            viewModel = appViewModel,
        )
    }
}
```

1.2.5 Single Module App-Core-Feature Organization

The project's source code is organized into app, core, and feature directories where app components are responsible for orchestrating cohesion between the features, core components are shared between features, and features refer to distinct chunks of functionality (e.g. different pages). All code belongs to a single Gradle module.

1.2.6 Repositories & Dependency Inversion

ClockWork abstracts data sources with interfaces called Repositories. The interfaces are defined in the Domain layer while their implementations are defined in the data layer. This principle is called Dependency Inversion and allows the UI layer to use Data layer components without depending on it, which would violate the Dependency Rule (see Clean Architecture).

1.2.7 Use Cases

Business logic (programming that addresses the purpose of the software solution) is encapsulated in units called Use Cases. Use Cases are given access to their collaborators at instantiation (see Dependency Injection) so that their dependents do not also need to depend on the Use Case's collaborators, reducing coupling. This also makes the Use Cases reusable. Use Cases are themselves collaborators and are injected into their dependents. Most Repository dependencies, particularly when the Repository serves multiple features, are encapsulated in Use Cases so that the other functions of the Repository are hidden from dependents that do not use them. These Use Cases are marked "Trivial Delegate" in this document.

```
class GetSessionUseCase(
    private val sessionRepository: TaskRepository
) {
    operator fun invoke(sessionId: Long): Flow<Task> {
        return sessionRepository.getTask(sessionId)
    }
}
```

1.2.8 Navigation

In an effort to reduce coupling, make the navigation system extensible, and keep feature logic isolated, each feature is responsible for defining its own navigation functions. There are two types of navigation functions: `NavGraph` registration and navigateToPage. Each feature has a navigation file in its top-level directory, wherein the page route and the two navigation functions are defined.

The page route is an `@Serializable` data class or object that is responsible for identifying the page in the `NavGraph` as well as transferring navigation arguments to the page.

The `NavGraph` registration function is responsible for creating the page while navigating, and is expressed as a `NavGraphBuilder` extension. The registration function propagates the page's navigation and `NavBar` dependencies to its parameters for the `NavHost` to provide. See ViewModel Instantiation for more information.
The navigateToPage function is expressed as a `NavController` extension and is a wrapper for `NavController.navigate()` with the route set as the page route.
The NavOption `launchSingleTop` is used to prevent multiple instances of the same page from being pushed to the `NavBackStack` if the user triggers the navigation repeatedly.

```
@Serializable
data class TimerRoute(val id: Long)

fun NavController.navigateToTimer(
   taskId: Long,
   navOptions: NavOptionsBuilder.() -> Unit = {
      launchSingleTop = true
   }
) {
   navigate(route = TimerRoute(taskId)) {
       navOptions()
   }
}
```

The `NavHost` at the top-level of the app hierarchy calls the `NavGraph` registration functions for all of the pages and provides the proper navigateToPage and navigate back functions to all the registration functions that require them using its `NavController`. The `NavHost` also instantiates the `NavBar`, which it also provides to the pages that need it.

```
val navBar = @Composable { currentDestination: Any ->
```

```
            NavBar(/* ... */)
}

sessionFormPage(
    onBackClick = navController::popBackStack,
    onCreateNewProfileClick = navController::navigateToCreateProfile,
)

taskListPage(
    navBar = { navBar(TaskListRoute) },
    onTaskClick = navController::navigateToTimer,
    onCreateNewTaskClick = navController::navigateToCreateNewSession,
)

timerPage(
    onBackClick = navController::popBackStack,
    onFinishClick = { sessionId ->
        navController.navigateToCompletion(sessionId) {
            popUpTo(route = TimerRoute(sessionId)) {
                inclusive = true
            }
        }
    },
    onEditClick = navController::navigateToEditSession,
)
```

1.2.9 State Hoisting

All pages in ClockWork implement the following pattern: a public composable function that is called in the `NavGraph` and receives high-level dependencies (e.g. the ViewModel, the `NavBar`, and other navigation closures) and a private composable function that actually contains the UI building code and receives a `uiState` structure and necessary closures. The wrapper collects the state from the ViewModel and "hoists" it down to the presenter. This makes the page previewable using the presenter composable, since it is stateless.

1.2.10 Permissions

Some Android features require permission from the user to be used. In ClockWork's case, the most notable of these is notifications. The compiler bundled with Android Studio lints for when these features are used without checking for permission, and the app will crash if it attempts to perform an action that requires permission without it being granted. Unfortunately, the system to request that the user grant permissions for the

app resides in the UI layer while the (very likely) the systems that need these permissions are in the domain layer. See the `PermissionRequestSignaller` for ClockWork's solution.

1.3 Disambiguation

Because of how the project evolved over time, component nomenclature is inconsistent.

1.3.1 Session/Task

Early work on the project only included functionality for timing instances of tasks (what are now called sessions). As such, the oldest components call Sessions Tasks e.g. `TaskRepository`, `Task`, `NewTask`, `StartedTask`, `CompletedTask`.

1.3.2 Profile/Template

A prevailing difficulty of the project is determining how to intuitively distinguish between the concept of a task and an instance of doing the task. The earliest solution was to name the former Profile and the latter Session; however, no one understood the concept and functionality of the Profile feature so it was changed to Template. No components use the name Template, it only appears in the user interface.

1.3.3 Composables/Elements

When the project was rearchitected to App-Core-Feature organization, each feature.ui folder had a composables folder for UI elements that appeared on the page. Sometimes that folder is named elements because the page itself is a Composable but does not appear in the composables folder.

1.3.4 SessionList/TodoList

The first feature developed was a list of sessions. When more lists of sessions were added, the purpose of the first list was changed to only show `NewTasks` and `StartedTasks` e.g. Todo Tasks.

1.3.5 SessionCompletion

The `session_completion_feature` refers to the page that shows the user their final times and estimation accuracy after they finish a session e.g. the Session Completion Report or Post Mortem Report or some such.

1.3.6 Running/Working

"Working" and "Running" are interchangeably used to refer to the timer tracking the time spent actively making progress towards a task's completion. While the timer does track the time spent on break (a brief distraction where returning to active work is imminent), this is not considered running.

1.3.7 Paused/On Break (and Contrasted with Suspended)

"Paused" and "On Break" are interchangeably used to refer to the timer tracking the time spent not working but intending to return to work soon. During this time, the user is not

performing a significant task but might be doing small activities, such as a bathroom or food break.
Paused and Suspended are **not** synonymous, the latter meaning that the user has concluded their effort on the given task for a significant period of time and has diverted their attention to another nontrivial task.

2. Components of Note

2.1 MainApplication

2.1.1 PermissionRequestSignaller

The `PermissionRequestSignaller` is the central component to the system that enables any component of the application to request special permissions from the system as necessary. The system uses a `Channel`, which enables components to send the request asynchronously then await a response from the handler, which is placed in the `MainActivity`. The handler collects emissions from the `Channel` and launches the `permissionLauncher` for each one. The `permissionLauncher` returns the result back to the original caller using a `CompletableDeferred` that was bundled with the permission sent in the `Channel`.

2.1.2 On First Launch Flow

"On First Launch Flow" refers to the system that detects the first launch of the app and the actions executed in that event. The components involved are: `MainActivity`, `MainViewModel`, and `AppPreferencesRepository`. The system is used to request permission to post notifications immediately when the user first opens the app and was implemented along with task reminders, since there was not a clearly appropriate place to request notification permission along a reminder's lifecycle.
The `AppPreferencesRepository` writes a value to app storage to remember that the app has been opened before. The `MainActivity`, which runs the on first launch behavior, uses the `MainViewModel` to access this value.

2.2 Model Objects

2.2.1 Task

The Task Interface holds all the necessary abstract variables necessary for any particular type of task.

2.2.1.1 NewTask

The NewTask class overrides the necessary values found in the Task interface for facilitating a new, unstarted task.

2.2.1.2 StartedTask

The StartedTask class overrides the necessary values found in the Task interface for an already started task with progress. This class also has extra functionality for proper working and break time, as well as status retrieval.

2.2.1.3 CompletedTask

The CompletedTaskClass overrides the necessary values found in the Task interface for storing a completed task. This class also handles solving for certain values.

2.2.2 Segment

The Segment class holds the structure for a segment of time within a particular task.

2.2.3 Marker

The Marker class holds the structure for a time marker found on a particular task.

2.2.4 Reminder

The Reminder class holds the structure for a reminder for a particular task.

2.2.5 Profile

The Profile class holds the structure for a Profile/Template.

2.3 Reminders

2.3.1 SessionReminderScheduler

The `SessionReminderScheduler` is responsible for creating and removing `ReminderWorker` entries from Android's `WorkManager` system as directed by other ClockWork components (in this case `CreateSessionUseCase` and `UpdateSessionUseCase` for `SessionReminderScheduler.schedule()`, and `CreateSessionUseCase`, `UpdateSessionUseCase` and `DeleteSessionUseCase` for `SessionReminderScheduler.cancelAllForSession()`, but the `SessionReminderScheduler` does not care). The `SessionReminderScheduler.schedule()` receives the reminder data as a parameter and repackages it into a `Work.Data` structure that is used to create the `ReminderWorker`. The `Work.Data` structure associates the data with ReminderWorker keys so the `ReminderWorker` can access them when it posts the notification.

2.3.2 ProcessScheduledReminderUseCase

The `ProcessScheduledReminderUseCase` encapsulates the `ReminderWorker`'s responsibility to post its reminder notification. When invoked, it checks to make sure that the reminder is within three minutes of its scheduled time before it posts it using the `ReminderNotificationManager`. It then marks the reminder record in the database complete or expired.

2.3.3 ReminderWorker

`ReminderWorker`'s responsibility is to represent a reminder in Android's work scheduling system and also to create the notification for the reminder when the scheduling system activates it. When activated, the reminder data is extracted from the input `Work.Data` structure using `ReminderWorker`'s defined keys. Then, `ProcessScheduledReminderUseCase` is invoked, which checks if the notification is late by five minutes; if so, the notification is not posted.

2.4 Reused Use Cases

2.4.1 GetAppEstimateUseCase

The `GetAppEstimateUseCase` uses user historical data to predict a range for a given session's duration given its properties. The entire calculation is based on applying an error to the user's estimate to result in a range of probable actual durations. First it calculates the similarity between the given session and each historical record using Gower's similarity. Each field to consider is defined in a `GowerField` structure which contains the equation to compare fields and a weight. The `fieldRange()` function determines the range of a field over the entire dataset. The following is an example for the difficulty field:

```
GowerField(
   similarityExpr = { s1, s2 ->
       val range = fieldRange(
           dataSet = sessionHistory.plus(todoSession)
       ) {
           it.difficulty
       }.let {
           val (min, max) = it
           max.minus(min)
       }

       // Range 0 mean all values are the same
       if (range == 0) return@GowerField 1.0

       s1.difficulty
           .minus(s2.difficulty)
           .absoluteValue
           .div(range.toDouble())
   },
   weight = 1.0
)
```

Next, a recency score is calculated for each record. Currently, the transformation is shaped like an inverted S curve and is configured to decay until the record's score is 0.5 after four months. The recency scores and similarity scores are pairwise combined with their own weights to control their contributions.

The error for each record is then calculated then averaged with the combined scores are the weights. Finally, the standard deviation of the dataset is calculated using the scores as weights as well. The high and low values are obtained with the average and standard deviation and then returned.

2.4.2 GetAverageEstimateErrorUseCase

The GetAverageEstimateErrorUseCase class gathers user data in order to generate the average estimate error of the user given a particular profile ID.

2.4.3 GetAverageSessionDurationUseCase

The GetAverageSessionDurationUseCase class gathers user data from the TaskRepository and utilizes linear regression to calculate the average length of a task provided a difficulty setting.

2.4.4 CalculateEstimateAccuracyUseCase

The `CalculateEstimateAccuracyUseCase` was written to make the user estimate accuracy calculation strategy reusable throughout the app in case it needs to be altered. Currently, it is 100% minus the canonical percent error calculation.

2.5 Timer

2.5.1 SessionTimer

The SessionTimer class offers all of the necessary tools and functionality for handling the different timers found in a session and is responsible for generating the elapsed time data to display in the app and notification.

2.5.2 TimerNotificationManager

The TimerNotificationManager creates a notification channel, builds the live notification that displays the state of the timer and has controls, and displays notification.

2.5.3 TimerNotificationActionProvider

The TimerNotificationActionProvider creates intents for interfacing with the TimerService. This class creates methods to be called by the on-notification buttons on the active timer notification.

2.5.4 CompleteStartedSessionUseCase

The CompleteStartedSessionUseCase class handles the transition from a started task to a completed one, properly capping off the final segment and stopping any notifications.

2.5.5 EndLastSegmentAndStartNewUseCase

The EndLastSegmentAndStartNewUseCase class simply ends the current segment for a task and creates and assigns a new one.

2.5.6 StartNewSessionUseCase

The StartNewSessionUseCase class handles the transition from a new task to a started one, creating the initial time segment.

2.5.7 TimerService

The `TimerService` is responsible for managing all Android service functionality so that the timer can persist independently of the app, loading the session data into the timer, and saving session progress. It also manages the `SessionTimer`, which is responsible for generating the elapsed time data to display in the app and notification.

2.5.8 TimerServiceIntentFactory

The `TimerServiceIntentFactory` class produces intents to be used by the application to invoke functions in the `TimerService`, including starting, pausing, resuming, breaking, completing, and adding a marker for a task.

2.5.9 TimerRepository

The `TimerRepository`'s responsibilities are to manage the connection to the `TimerService` and to enable communication between the `TimerService` and the rest of the system. The connection is established when initializing the `TimerRepository.state` flow property. Once established, the state exposed by the `TimerService` is observed and is reemitted through the `TimerRepository.state`. The `TimerRepository` uses `Intent`s to invoke `TimerService` functions, as is proper for Android services. The `Intent`s are created by the `TimerServiceIntentFactory` to improve `TimerRepository` cohesion.

2.6 Utility

2.6.1 FontScaling

Compose has two units that control the size of elements on the screen: `dp` and `sp`. `Dp` stands for density pixels and is designed to normalize screen sizes and resolutions. `Sp` stands for scalable pixels and adheres to the system text scaling settings. Text elements use `sp` while everything else uses `dp`. The `FontScaling` component contains two utility functions: `TextUnit.getScaling()` returns the current system font scale setting and `Int.dpScaledWith()` applies the current system font scaling to an int given reference `sp` value and returns it as a `dp`. `Int.dpScaledWith()` is used to make icons scale with their accompanying text.