Using Cursor for Mobile Development

Android + Kotlin + Jetpack Compose with AI →

Contact Info

Ken Kousen
Kousen IT, Inc.

- ken.kousen@kousenit.com
- http://www.kousenit.com
- http://kousenit.org (blog)
- Social Media:
  - @kenkousen (Twitter)
  - @kousenit.com (Bluesky)
  - https://www.linkedin.com/in/kenkousen/ (LinkedIn)
- *Tales from the jar side* (free newsletter)
  - https://kenkousen.substack.com
  - https://youtube.com/@talesfromthejarside

Course Overview: 5 Sessions

Course Overview: 5 Sessions

1. **Using Cursor for Java Development** (Session 1 - Complete)
   - Understanding code, navigation, generation, testing

Course Overview: 5 Sessions

1. **Using Cursor for Java Development** (Session 1 - Complete)
   - Understanding code, navigation, generation, testing
2. **Using Cursor for Mobile Development** (Today - 3 hours)
   - Android/Kotlin with AI assistance

Course Overview: 5 Sessions

1. **Using Cursor for Java Development** (Session 1 - Complete)
   - Understanding code, navigation, generation, testing
2. **Using Cursor for Mobile Development** (Today - 3 hours)
   - Android/Kotlin with AI assistance
3. **Agentic Coding with Cursor** (3 hours)
   - Advanced AI workflows and automation

Course Overview: 5 Sessions

1. **Using Cursor for Java Development** (Session 1 - Complete)
   - Understanding code, navigation, generation, testing
2. **Using Cursor for Mobile Development** (Today - 3 hours)
   - Android/Kotlin with AI assistance
3. **Agentic Coding with Cursor** (3 hours)
   - Advanced AI workflows and automation
4. **Reviewing and Testing Code** (3 hours)
   - Quality assurance with AI

Course Overview: 5 Sessions

1. **Using Cursor for Java Development** (Session 1 - Complete)
   - Understanding code, navigation, generation, testing
2. **Using Cursor for Mobile Development** (Today - 3 hours)
   - Android/Kotlin with AI assistance
3. **Agentic Coding with Cursor** (3 hours)
   - Advanced AI workflows and automation
4. **Reviewing and Testing Code** (3 hours)
   - Quality assurance with AI
5. **Exploring Agents and MCP** (3 hours)
   - Model Context Protocol and advanced features

Today’s Session: What We’ll Cover

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together
- **Cursor Composer Mode** - Multi-file code generation with `@codebase`

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together
- **Cursor Composer Mode** - Multi-file code generation with `@codebase`
- **Building with Compose** - UI components, state management, navigation

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together
- **Cursor Composer Mode** - Multi-file code generation with `@codebase`
- **Building with Compose** - UI components, state management, navigation
- **Data Persistence** - Room database with AI assistance

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together
- **Cursor Composer Mode** - Multi-file code generation with `@codebase`
- **Building with Compose** - UI components, state management, navigation
- **Data Persistence** - Room database with AI assistance
- **Testing** - ViewModel and Compose UI tests

Today’s Session: What We’ll Cover

- **The Hybrid Workflow** - Cursor + Android Studio working together
- **Cursor Composer Mode** - Multi-file code generation with `@codebase`
- **Building with Compose** - UI components, state management, navigation
- **Data Persistence** - Room database with AI assistance
- **Testing** - ViewModel and Compose UI tests
- **Production Patterns** - Hilt DI, Material 3 theming, accessibility

Two Projects Today

Two Projects Today

Project 1: Task Manager App

Two Projects Today

Project 1: Task Manager App

Build together from scratch - complete Android app with:

Two Projects Today

Project 1: Task Manager App

Build together from scratch - complete Android app with:

- Jetpack Compose UI
- ViewModels + StateFlow
- Navigation
- Room database

Two Projects Today

Project 1: Task Manager App

Build together from scratch - complete Android app with:

- Jetpack Compose UI
- ViewModels + StateFlow
- Navigation
- Room database

Project 2: Now in Android

Two Projects Today

Project 1: Task Manager App

Build together from scratch - complete Android app with:

- Jetpack Compose UI
- ViewModels + StateFlow
- Navigation
- Room database

Project 2: Now in Android

Explore Google’s production sample app with AI

Session 1 Recap

What You Learned

Today: Composer Mode

The Power Tool

Session 1 Recap

What You Learned

- **Chat Mode** (Cmd/Ctrl+L)
  - Ask questions
  - Understand code
  - Learn APIs

Today: Composer Mode

The Power Tool

Session 1 Recap

What You Learned

- **Chat Mode** (Cmd/Ctrl+L)
  - Ask questions
  - Understand code
  - Learn APIs
- **Agent Mode** (Cmd/Ctrl+I)
  - Generate code
  - Refactor
  - Single-file edits

Today: Composer Mode

The Power Tool

Session 1 Recap

What You Learned

- **Chat Mode** (Cmd/Ctrl+L)
  - Ask questions
  - Understand code
  - Learn APIs
- **Agent Mode** (Cmd/Ctrl+I)
  - Generate code
  - Refactor
  - Single-file edits

Today: Composer Mode

The Power Tool

- **Composer** (Cmd/Ctrl+Shift+I)
  - Multi-file generation
  - Full codebase context
  - Complex features
  - Iterative refinement

Session 1 Recap

What You Learned

- **Chat Mode** (Cmd/Ctrl+L)
  - Ask questions
  - Understand code
  - Learn APIs
- **Agent Mode** (Cmd/Ctrl+I)
  - Generate code
  - Refactor
  - Single-file edits

Today: Composer Mode

The Power Tool

- **Composer** (Cmd/Ctrl+Shift+I)
  - Multi-file generation
  - Full codebase context
  - Complex features
  - Iterative refinement

Tag `@codebase` for AI to see everything

Part 1: The Hybrid Workflow

Why You Need BOTH Tools

Android Studio

Cursor

Android Studio

✅ **Great for:**

Cursor

Android Studio

✅ **Great for:**

- Project creation
- Running apps
- Emulator/device management
- Visual layout preview
- Debugging
- Build management
- SDK management

Cursor

Android Studio

✅ **Great for:**

- Project creation
- Running apps
- Emulator/device management
- Visual layout preview
- Debugging
- Build management
- SDK management

Cursor

✅ **Great for:**

Android Studio

✅ **Great for:**

- Project creation
- Running apps
- Emulator/device management
- Visual layout preview
- Debugging
- Build management
- SDK management

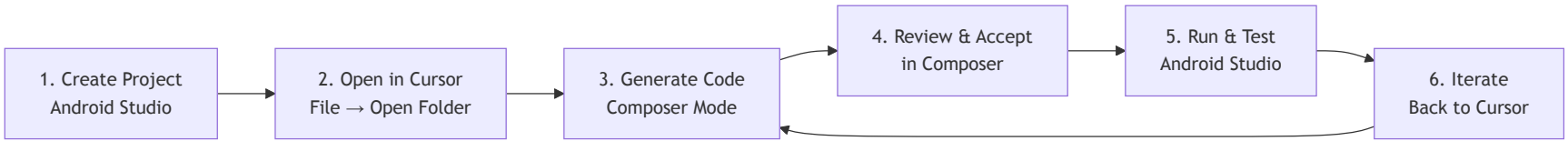
Cursor

✅ **Great for:**

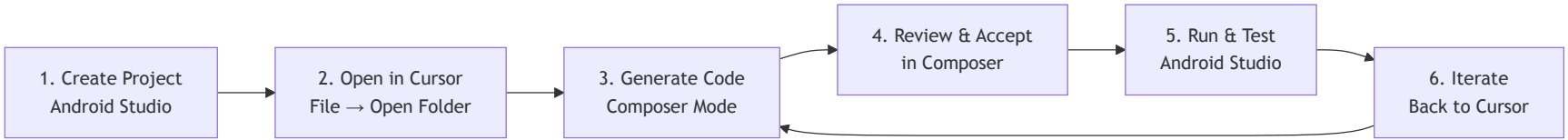
- AI-powered code generation
- Understanding APIs
- Refactoring
- Test generation
- Boilerplate elimination
- Architecture guidance

The Efficient Workflow

The Efficient Workflow



The Efficient Workflow



Key: Use each tool for what it's best at!

Cursor Composer Mode

Your New Superpower

Cursor Composer Mode

Your New Superpower

Open Composer:

Cursor Composer Mode

Your New Superpower

Open Composer:

- Mac: `Cmd + Shift + I`
- Windows/Linux: `Ctrl + Shift + I`

Cursor Composer Mode

Your New Superpower

Open Composer:

- Mac: `Cmd + Shift + I`
- Windows/Linux: `Ctrl + Shift + I`

Best Practices:

Cursor Composer Mode

Your New Superpower

Open Composer:

- Mac: `Cmd + Shift + I`
- Windows/Linux: `Ctrl + Shift + I`

Best Practices:

```
@codebase Create a TaskCard composable using Material 3
components with title, description, and completion checkbox.
Use proper spacing and modern Compose patterns.
```

Cursor Composer Mode

Your New Superpower

Open Composer:

- Mac: `Cmd + Shift + I`
- Windows/Linux: `Ctrl + Shift + I`

Best Practices:

```
@codebase Create a TaskCard composable using Material 3
components with title, description, and completion checkbox.
Use proper spacing and modern Compose patterns.
```

- Always tag `@codebase` for context
- Be specific and detailed
- Describe architecture patterns
- Mention dependencies if needed

Demo: Setting Up the Hybrid Workflow

Demo: Setting Up the Hybrid Workflow

1. **Create project in Android Studio**
   - File → New → New Project → Empty Activity
   - Wait for Gradle sync

Demo: Setting Up the Hybrid Workflow

1. **Create project in Android Studio**
   - File → New → New Project → Empty Activity
   - Wait for Gradle sync
2. **Open same folder in Cursor**
   - File → Open Folder
   - Navigate to Android project

Demo: Setting Up the Hybrid Workflow

1. **Create project in Android Studio**
   - File → New → New Project → Empty Activity
   - Wait for Gradle sync
2. **Open same folder in Cursor**
   - File → Open Folder
   - Navigate to Android project
3. **Test Composer in Cursor**
   - `Cmd/Ctrl+Shift+I`
   - `@codebase Explain this project structure`

Demo: Setting Up the Hybrid Workflow

1. **Create project in Android Studio**
   - File → New → New Project → Empty Activity
   - Wait for Gradle sync
2. **Open same folder in Cursor**
   - File → Open Folder
   - Navigate to Android project
3. **Test Composer in Cursor**
   - `Cmd/Ctrl+Shift+I`
   - `@codebase Explain this project structure`
4. **Back to Android Studio**
   - Run app (Shift+F10)

Exercise: Environment Check

You should have:

Exercise: Environment Check

You should have:

- ✅ Android Studio installed and running
- ✅ At least one emulator configured (or physical device)
- ✅ Cursor installed
- ✅ Same Android project open in both tools
- ✅ Composer mode tested (`Cmd/Ctrl+Shift+I`)

Exercise: Environment Check

You should have:

- ✅ Android Studio installed and running
- ✅ At least one emulator configured (or physical device)
- ✅ Cursor installed
- ✅ Same Android project open in both tools
- ✅ Composer mode tested (`Cmd/Ctrl+Shift+I`)

Not working? Ask for help now!

Part 2: Building UIs with Jetpack Compose

From Zero to UI Components

Jetpack Compose Basics

Jetpack Compose Basics

Declarative UI

Jetpack Compose Basics

Declarative UI

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Jetpack Compose Basics

Declarative UI

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Key Concepts

Jetpack Compose Basics

Declarative UI

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Key Concepts

- Functions, not XML
- Recomposition on state change
- Material 3 components
- Modifiers for styling

Demo: Generating a TaskCard

In Cursor Composer (Cmd/Ctrl+Shift+I):

```
@codebase Create a TaskCard composable that displays a task
with title, description, and completed checkbox using
Material 3 components. Use proper spacing and modern
Compose patterns.
```

Demo: Generating a TaskCard

In Cursor Composer (Cmd/Ctrl+Shift+I):

```
@codebase Create a TaskCard composable that displays a task
with title, description, and completed checkbox using
Material 3 components. Use proper spacing and modern
Compose patterns.
```

AI will generate:

Demo: Generating a TaskCard

In Cursor Composer (Cmd/Ctrl+Shift+I):

```
@codebase Create a TaskCard composable that displays a task
with title, description, and completed checkbox using
Material 3 components. Use proper spacing and modern
Compose patterns.
```

AI will generate:

- `@Composable` function
- Material 3 Card, Text, Checkbox
- Proper modifiers (padding, fillMaxWidth)
- State handling

Demo: Generating a TaskCard

In Cursor Composer (Cmd/Ctrl+Shift+I):

```
@codebase Create a TaskCard composable that displays a task
with title, description, and completed checkbox using
Material 3 components. Use proper spacing and modern
Compose patterns.
```

AI will generate:

- `@Composable` function
- Material 3 Card, Text, Checkbox
- Proper modifiers (padding, fillMaxWidth)
- State handling

Accept changes → Switch to Android Studio → Run

Preview Functions

Back in Cursor Composer:

```
@codebase Add preview functions showing TaskCard with
different states: uncompleted task, completed task,
and long text overflow
```

Preview Functions

Back in Cursor Composer:

```
@codebase Add preview functions showing TaskCard with
different states: uncompleted task, completed task,
and long text overflow
```

Why previews matter:

Preview Functions

Back in Cursor Composer:

```
@codebase Add preview functions showing TaskCard with
different states: uncompleted task, completed task,
and long text overflow
```

Why previews matter:

- See UI without running app
- Test different states
- Faster iteration

Preview Functions

Back in Cursor Composer:

```
@codebase Add preview functions showing TaskCard with
different states: uncompleted task, completed task,
and long text overflow
```

Why previews matter:

- See UI without running app
- Test different states
- Faster iteration

View in Android Studio's preview pane

Material Design 3

Material Design 3

Key Components

Material Design 3

Key Components

- **Card** - Container for related content
- **Text** - Typography with styles
- **Button** - Primary, Secondary, Text
- **Icon** - Material icons
- **TextField** - Input fields

Material Design 3

Key Components

- **Card** - Container for related content
- **Text** - Typography with styles
- **Button** - Primary, Secondary, Text
- **Icon** - Material icons
- **TextField** - Input fields

Styling

Material Design 3

Key Components

- **Card** - Container for related content
- **Text** - Typography with styles
- **Button** - Primary, Secondary, Text
- **Icon** - Material icons
- **TextField** - Input fields

Styling

- **Modifiers** - padding, size, layout
- **Colors** - From theme
- **Typography** - Headlines, body, labels

Demo: Building a TaskList

In Cursor Composer:

```
@codebase Create a TaskList composable using LazyColumn
to display a list of tasks. Include proper keys and
content types. Handle empty state. Add sample data for preview.
```

Demo: Building a TaskList

In Cursor Composer:

```
@codebase Create a TaskList composable using LazyColumn
to display a list of tasks. Include proper keys and
content types. Handle empty state. Add sample data for preview.
```

AI generates:

Demo: Building a TaskList

In Cursor Composer:

```
@codebase Create a TaskList composable using LazyColumn
to display a list of tasks. Include proper keys and
content types. Handle empty state. Add sample data for preview.
```

AI generates:

- LazyColumn for efficient scrolling
- Item keys for performance
- Empty state handling
- Sample data

Demo: Building a TaskList

In Cursor Composer:

```
@codebase Create a TaskList composable using LazyColumn
to display a list of tasks. Include proper keys and
content types. Handle empty state. Add sample data for preview.
```

AI generates:

- LazyColumn for efficient scrolling
- Item keys for performance
- Empty state handling
- Sample data

Run in Android Studio

LazyColumn Essentials

```
@Composable
fun TaskList(tasks: List<Task>) {
    LazyColumn {
        items(
            items = tasks,
            key = { task -> task.id }
        ) { task ->
            TaskCard(task = task)
        }
    }
}
```

LazyColumn Essentials

```
@Composable
fun TaskList(tasks: List<Task>) {
    LazyColumn {
        items(
            items = tasks,
            key = { task -> task.id }
        ) { task ->
            TaskCard(task = task)
        }
    }
}
```

- **LazyColumn** - Like RecyclerView, but simpler

LazyColumn Essentials

```
@Composable
fun TaskList(tasks: List<Task>) {
    LazyColumn {
        items(
            items = tasks,
            key = { task -> task.id }
        ) { task ->
            TaskCard(task = task)
        }
    }
}
```

- **LazyColumn** - Like RecyclerView, but simpler
- **items()** - List of items

LazyColumn Essentials

```
@Composable
fun TaskList(tasks: List<Task>) {
    LazyColumn {
        items(
            items = tasks,
            key = { task -> task.id }
        ) { task ->
            TaskCard(task = task)
        }
    }
}
```

- **LazyColumn** - Like RecyclerView, but simpler
- **items()** - List of items
- **key** - Performance optimization

LazyColumn Essentials

```
@Composable
fun TaskList(tasks: List<Task>) {
    LazyColumn {
        items(
            items = tasks,
            key = { task -> task.id }
        ) { task ->
            TaskCard(task = task)
        }
    }
}
```

- **LazyColumn** - Like RecyclerView, but simpler
- **items()** - List of items
- **key** - Performance optimization
- Automatically handles scrolling

Exercise: Build Your UI

In Cursor Composer:

Exercise: Build Your UI

In Cursor Composer:

1. Generate a customized TaskCard
2. Add preview functions for different states
3. Create a TaskList with sample data
4. Use Chat mode: "How do I make the checkbox larger?"
5. Apply changes in Composer
6. Test in Android Studio

Exercise: Build Your UI

In Cursor Composer:

1. Generate a customized TaskCard
2. Add preview functions for different states
3. Create a TaskList with sample data
4. Use Chat mode: "How do I make the checkbox larger?"
5. Apply changes in Composer
6. Test in Android Studio

Time: 15 minutes

Part 3: State Management & Architecture

MVVM with ViewModels

Understanding MVVM

First, ask Chat Mode:

```
Cmd/Ctrl+L: "Explain the MVVM pattern for Android
and how it applies to Jetpack Compose"
```

Understanding MVVM

First, ask Chat Mode:

```
Cmd/Ctrl+L: "Explain the MVVM pattern for Android
and how it applies to Jetpack Compose"
```

Model - Data and business logic **View** - UI (Composables) **ViewModel** - State and events

Understanding MVVM

First, ask Chat Mode:

```
Cmd/Ctrl+L: "Explain the MVVM pattern for Android
and how it applies to Jetpack Compose"
```

Model - Data and business logic **View** - UI (Composables) **ViewModel** - State and events

Unidirectional Data Flow:

Understanding MVVM

First, ask Chat Mode:

```
Cmd/Ctrl+L: "Explain the MVVM pattern for Android
and how it applies to Jetpack Compose"
```

Model - Data and business logic **View** - UI (Composables) **ViewModel** - State and events

Unidirectional Data Flow:

- State flows down
- Events flow up

Demo: Creating a ViewModel

In Cursor Composer:

```
@codebase Create a TaskViewModel that manages a list of
tasks using StateFlow. Include functions to add, update,
delete, and toggle task completion. Use proper coroutine
scopes and follow Android best practices.
```

Demo: Creating a ViewModel

In Cursor Composer:

```
@codebase Create a TaskViewModel that manages a list of
tasks using StateFlow. Include functions to add, update,
delete, and toggle task completion. Use proper coroutine
scopes and follow Android best practices.
```

AI generates:

Demo: Creating a ViewModel

In Cursor Composer:

```
@codebase Create a TaskViewModel that manages a list of
tasks using StateFlow. Include functions to add, update,
delete, and toggle task completion. Use proper coroutine
scopes and follow Android best practices.
```

AI generates:

- StateFlow for state
- MutableStateFlow internally
- viewModelScope for coroutines
- Immutable state updates

StateFlow Pattern

```
class TaskViewModel : ViewModel() {
    private val _tasks = MutableStateFlow<List<Task>>(emptyList())
    val tasks: StateFlow<List<Task>> = _tasks.asStateFlow()

    fun addTask(task: Task) {
        _tasks.value = _tasks.value + task
    }

    fun toggleCompletion(taskId: String) {
        _tasks.value = _tasks.value.map { task ->
            if (task.id == taskId) {
                task.copy(completed = !task.completed)
            } else task
        }
    }
}
```

UI State Pattern

In Cursor Composer:

```
@codebase Create a sealed interface TaskListUiState with
states for Loading, Success with task list, and Error with message
```

UI State Pattern

In Cursor Composer:

```
@codebase Create a sealed interface TaskListUiState with
states for Loading, Success with task list, and Error with message
```

```
sealed interface TaskListUiState {
    data object Loading : TaskListUiState
    data class Success(val tasks: List<Task>) : TaskListUiState
    data class Error(val message: String) : TaskListUiState
}
```

UI State Pattern

In Cursor Composer:

```
@codebase Create a sealed interface TaskListUiState with
states for Loading, Success with task list, and Error with message
```

```
sealed interface TaskListUiState {
    data object Loading : TaskListUiState
    data class Success(val tasks: List<Task>) : TaskListUiState
    data class Error(val message: String) : TaskListUiState
}
```

Why sealed interfaces?

UI State Pattern

In Cursor Composer:

```
@codebase Create a sealed interface TaskListUiState with
states for Loading, Success with task list, and Error with message
```

```
sealed interface TaskListUiState {
    data object Loading : TaskListUiState
    data class Success(val tasks: List<Task>) : TaskListUiState
    data class Error(val message: String) : TaskListUiState
}
```

Why sealed interfaces?

- Type-safe state representation
- Exhaustive when expressions
- Clear state transitions

Connecting ViewModel to UI

In Cursor Composer:

```
@codebase Update TaskList composable to observe
TaskViewModel's StateFlow using collectAsStateWithLifecycle
and handle all UI states (Loading, Success, Error)
```

Connecting ViewModel to UI

In Cursor Composer:

```
@codebase Update TaskList composable to observe
TaskViewModel's StateFlow using collectAsStateWithLifecycle
and handle all UI states (Loading, Success, Error)
```

```
@Composable
fun TaskList(viewModel: TaskViewModel = viewModel()) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    when (uiState) {
        is TaskListUiState.Loading -> LoadingIndicator()
        is TaskListUiState.Success -> TaskList(tasks)
        is TaskListUiState.Error -> ErrorMessage(message)
    }
}
```

Exercise: Implement State Management

Using Composer:

Exercise: Implement State Management

Using Composer:

1. Generate a TaskViewModel with StateFlow
2. Create a TaskListUiState sealed interface
3. Connect ViewModel to your TaskList composable
4. Run in Android Studio
5. Add a task and observe UI update
6. Test loading and error states

Exercise: Implement State Management

Using Composer:

1. Generate a TaskViewModel with StateFlow
2. Create a TaskListUiState sealed interface
3. Connect ViewModel to your TaskList composable
4. Run in Android Studio
5. Add a task and observe UI update
6. Test loading and error states

Time: 10 minutes

Break Time! ☕

10 Minutes

Stretch, grab coffee, be back on time!

Part 4: Navigation

Building Multi-Screen Apps

Compose Navigation Basics

Compose Navigation Basics

Key Concepts

Compose Navigation Basics

Key Concepts

- **NavController** - Manages navigation
- **NavHost** - Container for destinations
- **Routes** - String identifiers for screens
- **Arguments** - Pass data between screens

Compose Navigation Basics

Key Concepts

- **NavController** - Manages navigation
- **NavHost** - Container for destinations
- **Routes** - String identifiers for screens
- **Arguments** - Pass data between screens

Type-Safe Navigation

Compose Navigation Basics

Key Concepts

- **NavController** - Manages navigation
- **NavHost** - Container for destinations
- **Routes** - String identifiers for screens
- **Arguments** - Pass data between screens

Type-Safe Navigation

Use sealed classes for routes with parameters

Demo: Setting Up Navigation

In Cursor Composer:

```
@codebase Add Jetpack Compose Navigation to this project.
Create a sealed class for routes: HomeRoute and
TaskDetailRoute with task ID parameter. Add the required
dependencies to build.gradle if needed.
```

Demo: Setting Up Navigation

In Cursor Composer:

```
@codebase Add Jetpack Compose Navigation to this project.
Create a sealed class for routes: HomeRoute and
TaskDetailRoute with task ID parameter. Add the required
dependencies to build.gradle if needed.
```

AI will:

Demo: Setting Up Navigation

In Cursor Composer:

```
@codebase Add Jetpack Compose Navigation to this project.
Create a sealed class for routes: HomeRoute and
TaskDetailRoute with task ID parameter. Add the required
dependencies to build.gradle if needed.
```

AI will:

- Add navigation dependencies
- Create Route sealed class
- Set up type-safe navigation

Demo: Setting Up Navigation

In Cursor Composer:

```
@codebase Add Jetpack Compose Navigation to this project.
Create a sealed class for routes: HomeRoute and
TaskDetailRoute with task ID parameter. Add the required
dependencies to build.gradle if needed.
```

AI will:

- Add navigation dependencies
- Create Route sealed class
- Set up type-safe navigation

Build in Android Studio

Navigation Routes

```
sealed class Route(val route: String) {
    data object Home : Route("home")
    data class TaskDetail(val taskId: String) : Route("task/{taskId}") {
        companion object {
            const val TASK_ID_KEY = "taskId"
        }
    }
}
```

Navigation Routes

```
sealed class Route(val route: String) {
    data object Home : Route("home")
    data class TaskDetail(val taskId: String) : Route("task/{taskId}") {
        companion object {
            const val TASK_ID_KEY = "taskId"
        }
    }
}
```

Benefits:

Navigation Routes

```
sealed class Route(val route: String) {
    data object Home : Route("home")
    data class TaskDetail(val taskId: String) : Route("task/{taskId}") {
        companion object {
            const val TASK_ID_KEY = "taskId"
        }
    }
}
```

Benefits:

- Type-safe
- Compile-time checking
- Refactoring-friendly

Creating NavHost

In Cursor Composer:

```
@codebase Create a NavHost with home screen showing task
list and detail screen for editing a task. Set up the
navigation structure in MainActivity.
```

Creating NavHost

In Cursor Composer:

```
@codebase Create a NavHost with home screen showing task
list and detail screen for editing a task. Set up the
navigation structure in MainActivity.
```

```
@Composable
fun TaskNavHost(navController: NavHostController) {
    NavHost(navController, startDestination = Route.Home.route) {
        composable(Route.Home.route) {
            TaskListScreen(onTaskClick = { taskId ->
                navController.navigate(Route.TaskDetail(taskId).route)
            })
        }
        composable(
            route = Route.TaskDetail.route,
            arguments = listOf(navArgument("taskId") { type = NavType.StringType })
        ) { backStackEntry ->
            val taskId = backStackEntry.arguments?.getString("taskId")
            TaskDetailScreen(taskId = taskId, onNavigateBack = {
                navController.popBackStack()
            })
        }
```

Demo: Detail Screen with Form

In Cursor Composer:

```
@codebase Create TaskDetailScreen composable that takes a
task ID, loads task from ViewModel, and provides form inputs
to edit title and description with Save and Cancel buttons.
Wire up navigation.
```

Demo: Detail Screen with Form

In Cursor Composer:

```
@codebase Create TaskDetailScreen composable that takes a
task ID, loads task from ViewModel, and provides form inputs
to edit title and description with Save and Cancel buttons.
Wire up navigation.
```

AI generates:

Demo: Detail Screen with Form

In Cursor Composer:

```
@codebase Create TaskDetailScreen composable that takes a
task ID, loads task from ViewModel, and provides form inputs
to edit title and description with Save and Cancel buttons.
Wire up navigation.
```

AI generates:

- Form with TextFields
- State management
- Save/Cancel logic
- Navigation handling

Demo: Detail Screen with Form

In Cursor Composer:

```
@codebase Create TaskDetailScreen composable that takes a
task ID, loads task from ViewModel, and provides form inputs
to edit title and description with Save and Cancel buttons.
Wire up navigation.
```

AI generates:

- Form with TextFields
- State management
- Save/Cancel logic
- Navigation handling

Run and test navigation flow

Exercise: Implement Navigation

Test your app:

Exercise: Implement Navigation

Test your app:

1. Click a task in the list
2. Navigate to detail screen
3. Edit task title and description
4. Click Save
5. Navigate back to list
6. Verify changes persist

Exercise: Implement Navigation

Test your app:

1. Click a task in the list
2. Navigate to detail screen
3. Edit task title and description
4. Click Save
5. Navigate back to list
6. Verify changes persist

Use Composer to fix any issues!

Exercise: Implement Navigation

Test your app:

1. Click a task in the list
2. Navigate to detail screen
3. Edit task title and description
4. Click Save
5. Navigate back to list
6. Verify changes persist

Use Composer to fix any issues!

Time: 5 minutes

Part 5: Data Persistence with Room

Making Data Last

Room Database Overview

Room Database Overview

Components

Room Database Overview

Components

- **Entity** - Database table (data class with @Entity)
- **DAO** - Data Access Object (interface with @Dao)
- **Database** - Singleton database instance

Room Database Overview

Components

- **Entity** - Database table (data class with @Entity)
- **DAO** - Data Access Object (interface with @Dao)
- **Database** - Singleton database instance

Benefits

Room Database Overview

Components

- **Entity** - Database table (data class with @Entity)
- **DAO** - Data Access Object (interface with @Dao)
- **Database** - Singleton database instance

Benefits

- Type-safe SQL
- Compile-time verification
- Flow support for reactive queries
- Coroutines integration

Demo: Creating Entity

In Cursor Composer:

```
@codebase Create a Task entity for Room database with fields:
id (auto-generated), title (not null), description (nullable),
completed (boolean), createdAt (timestamp). Add Room
dependencies to build.gradle if needed.
```

Demo: Creating Entity

In Cursor Composer:

```
@codebase Create a Task entity for Room database with fields:
id (auto-generated), title (not null), description (nullable),
completed (boolean), createdAt (timestamp). Add Room
dependencies to build.gradle if needed.
```

```
@Entity(tableName = "tasks")
data class Task(
    @PrimaryKey @ColumnInfo(name = "id")
    val id: String = UUID.randomUUID().toString(),

    @ColumnInfo(name = "title")
    val title: String,

    @ColumnInfo(name = "description")
    val description: String? = null,

    @ColumnInfo(name = "completed")
    val completed: Boolean = false,

    @ColumnInfo(name = "created_at")
    val createdAt: Long = System.currentTimeMillis()
)
```

Demo: Creating DAO

In Cursor Composer:

```
@codebase Create TaskDao with functions to: insert, update,
delete task, get all tasks as Flow, get task by ID, and get
completed/incomplete tasks. Use proper suspend functions and Flow.
```

Demo: Creating DAO

In Cursor Composer:

```
@codebase Create TaskDao with functions to: insert, update,
delete task, get all tasks as Flow, get task by ID, and get
completed/incomplete tasks. Use proper suspend functions and Flow.
```

```kotlin
@Dao
interface TaskDao {
    @Query("SELECT * FROM tasks ORDER BY created_at DESC")
    fun getAllTasks(): Flow<List<Task>>

    @Query("SELECT * FROM tasks WHERE id = :taskId")
    suspend fun getTaskById(taskId: String): Task?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    // ... other operations
}
```

Demo: Creating DAO

In Cursor Composer:

```
@codebase Create TaskDao with functions to: insert, update,
delete task, get all tasks as Flow, get task by ID, and get
completed/incomplete tasks. Use proper suspend functions and Flow.
```

```kotlin
@Dao
interface TaskDao {
    @Query("SELECT * FROM tasks ORDER BY created_at DESC")
    fun getAllTasks(): Flow<List<Task>>

    @Query("SELECT * FROM tasks WHERE id = :taskId")
    suspend fun getTaskById(taskId: String): Task?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(task: Task)

    @Update
    suspend fun update(task: Task)

    // ... other operations
}
```

Key patterns: Flow for reactive queries, suspend for one-shot operations

Demo: Creating Database

In Cursor Composer:

```
@codebase Create AppDatabase extending RoomDatabase with
TaskDao. Include version, entities, and proper database
configuration.
```

Demo: Creating Database

In Cursor Composer:

```
@codebase Create AppDatabase extending RoomDatabase with
TaskDao. Include version, entities, and proper database
configuration.
```

```
@Database(entities = [Task::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao
}
```

Demo: Creating Database (Singleton Pattern)

Demo: Creating Database (Singleton Pattern)

```
abstract class AppDatabase : RoomDatabase() {
    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(
                    context,
                    AppDatabase::class.java,
                    "app-db"
                ).fallbackToDestructiveMigration().build()
            }
    }
}
```

Demo: Creating Database (Singleton Pattern)

```
abstract class AppDatabase : RoomDatabase() {
    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(
                    context,
                    AppDatabase::class.java,
                    "app-db"
                ).fallbackToDestructiveMigration().build()
            }
    }
}
```

Pattern: Singleton with double-checked locking

Repository Pattern

In Cursor Composer:

```
@codebase Create TaskRepository that uses TaskDao and
exposes Flow for task list and suspend functions for CRUD
operations. Update TaskViewModel to use this real repository
instead of fake data.
```

Repository Pattern

In Cursor Composer:

```
@codebase Create TaskRepository that uses TaskDao and
exposes Flow for task list and suspend functions for CRUD
operations. Update TaskViewModel to use this real repository
instead of fake data.
```

Why Repository?

Repository Pattern

In Cursor Composer:

```
@codebase Create TaskRepository that uses TaskDao and
exposes Flow for task list and suspend functions for CRUD
operations. Update TaskViewModel to use this real repository
instead of fake data.
```

Why Repository?

- Abstraction over data source
- Easier testing
- Single source of truth
- Can combine local + remote data

Demo: Testing Persistence

In Android Studio:

Demo: Testing Persistence

In Android Studio:

1. Run the app
2. Add several tasks
3. Toggle some as completed
4. Force stop the app (don’t just background)
5. Reopen the app
6. **Tasks should still be there!** ✅

Demo: Testing Persistence

In Android Studio:

1. Run the app
2. Add several tasks
3. Toggle some as completed
4. Force stop the app (don’t just background)
5. Reopen the app
6. **Tasks should still be there!** ✅

If not, check:

Demo: Testing Persistence

In Android Studio:

1. Run the app
2. Add several tasks
3. Toggle some as completed
4. Force stop the app (don’t just background)
5. Reopen the app
6. **Tasks should still be there!** ✅

If not, check:

- Room dependencies added?
- Database initialized?
- ViewModel using real repository?

Exercise: Verify Persistence

Test in Android Studio:

Exercise: Verify Persistence

Test in Android Studio:

1. Add 3-5 tasks with different content
2. Complete some tasks
3. Close app completely
4. Reopen app
5. Verify all tasks persist
6. Edit a task and verify changes save

Exercise: Verify Persistence

Test in Android Studio:

1. Add 3-5 tasks with different content
2. Complete some tasks
3. Close app completely
4. Reopen app
5. Verify all tasks persist
6. Edit a task and verify changes save

Use Chat mode if issues arise!

Exercise: Verify Persistence

Test in Android Studio:

1. Add 3-5 tasks with different content
2. Complete some tasks
3. Close app completely
4. Reopen app
5. Verify all tasks persist
6. Edit a task and verify changes save

Use Chat mode if issues arise!

Time: 5 minutes

Break Time! ☕

10 Minutes

We’re past the halfway point!

Part 6: Testing Android Components

TDD with AI Assistance

Testing Philosophy

Testing Philosophy

Types of Tests

Testing Philosophy

Types of Tests

- **Unit Tests** - ViewModels, repositories (fast, isolated)
- **UI Tests** - Composables with ComposeTestRule
- **Integration Tests** - Multiple components together

Testing Philosophy

Types of Tests

- **Unit Tests** - ViewModels, repositories (fast, isolated)
- **UI Tests** - Composables with ComposeTestRule
- **Integration Tests** - Multiple components together

With AI

Testing Philosophy

Types of Tests

- **Unit Tests** - ViewModels, repositories (fast, isolated)
- **UI Tests** - Composables with ComposeTestRule
- **Integration Tests** - Multiple components together

With AI

- Generate comprehensive test suites quickly
- Cover edge cases you might miss
- Learn testing patterns

Testing Philosophy

Types of Tests

- **Unit Tests** - ViewModels, repositories (fast, isolated)
- **UI Tests** - Composables with ComposeTestRule
- **Integration Tests** - Multiple components together

With AI

- Generate comprehensive test suites quickly
- Cover edge cases you might miss
- Learn testing patterns

But always review generated tests!

Demo: ViewModel Testing

In Cursor Composer:

```
@codebase Generate unit tests for TaskViewModel using
JUnit 5 and MockK. Test adding a task, deleting a task,
and toggling completion. Use Turbine for testing Flow
emissions. Add test dependencies to build.gradle if needed.
```

Demo: ViewModel Testing

In Cursor Composer:

```
@codebase Generate unit tests for TaskViewModel using
JUnit 5 and MockK. Test adding a task, deleting a task,
and toggling completion. Use Turbine for testing Flow
emissions. Add test dependencies to build.gradle if needed.
```

AI generates:

Demo: ViewModel Testing

In Cursor Composer:

```
@codebase Generate unit tests for TaskViewModel using
JUnit 5 and MockK. Test adding a task, deleting a task,
and toggling completion. Use Turbine for testing Flow
emissions. Add test dependencies to build.gradle if needed.
```

AI generates:

- Test class structure
- Mock repository
- Coroutine test setup
- Flow testing with Turbine
- Multiple test cases

Demo: ViewModel Testing

In Cursor Composer:

```
@codebase Generate unit tests for TaskViewModel using
JUnit 5 and MockK. Test adding a task, deleting a task,
and toggling completion. Use Turbine for testing Flow
emissions. Add test dependencies to build.gradle if needed.
```

AI generates:

- Test class structure
- Mock repository
- Coroutine test setup
- Flow testing with Turbine
- Multiple test cases

Run: `./gradlew test`

ViewModel Test Example

```
@OptIn(ExperimentalCoroutinesApi::class)
class TaskViewModelTest {
    @get:Rule
    val mainDispatcherRule = MainDispatcherRule()

    private lateinit var repository: TaskRepository
    private lateinit var viewModel: TaskViewModel

    @Test
    fun `addTask updates state with new task`() = runTest {
        // Given
        coEvery { repository.insertTask(any()) } just Runs

        // When
        viewModel.addTask("Test Task", "Description")

        // Then
        viewModel.tasks.test {
            assertThat(awaitItem()).contains(newTask)
        }
    }
}
```

Demo: Compose UI Testing

In Cursor Composer:

```
@codebase Generate Compose UI tests for TaskCard using
ComposeTestRule. Test that title and description are
displayed, checkbox reflects completed state, and clicking
checkbox triggers callback.
```

Demo: Compose UI Testing

In Cursor Composer:

```
@codebase Generate Compose UI tests for TaskCard using
ComposeTestRule. Test that title and description are
displayed, checkbox reflects completed state, and clicking
checkbox triggers callback.
```

AI generates:

Demo: Compose UI Testing

In Cursor Composer:

```
@codebase Generate Compose UI tests for TaskCard using
ComposeTestRule. Test that title and description are
displayed, checkbox reflects completed state, and clicking
checkbox triggers callback.
```

AI generates:

- ComposeTestRule setup
- Finding nodes by semantic properties
- Assertions (text, state)
- User interactions (clicks)

Compose UI Test Example

```
class TaskCardTest {
    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun taskCard_displaysTaskInformation() {
        val task = Task(title = "Test Task", description = "Test Description")

        composeTestRule.setContent {
            TaskCard(task = task, onToggleComplete = {})
        }

        composeTestRule.onNodeWithText("Test Task").assertIsDisplayed()
        composeTestRule.onNodeWithText("Test Description").assertIsDisplayed()
    }
}
```

Key concepts: setContent, onNodeWithText, assertions

Semantic Properties

```
@Composable
fun TaskCard(task: Task, onToggleComplete: () -> Unit) {
    Card {
        Row(modifier = Modifier.semantics { testTag = "task-card" }) {
            Text(
                text = task.title,
                modifier = Modifier.semantics { testTag = "task-title" }
            )
        }
    }
}
```

Semantic Properties

```
@Composable
fun TaskCard(task: Task, onToggleComplete: () -> Unit) {
    Card {
        Row(modifier = Modifier.semantics { testTag = "task-card" }) {
            Text(
                text = task.title,
                modifier = Modifier.semantics { testTag = "task-title" }
            )
        }
    }
}
```

Semantic properties help with:

Semantic Properties

```
@Composable
fun TaskCard(task: Task, onToggleComplete: () -> Unit) {
    Card {
        Row(modifier = Modifier.semantics { testTag = "task-card" }) {
            Text(
                text = task.title,
                modifier = Modifier.semantics { testTag = "task-title" }
            )
        }
    }
}
```

Semantic properties help with:

- Testing
- Accessibility
- UI automation

Semantic Properties (Checkbox)

```
@Composable
fun TaskCheckbox(completed: Boolean, onToggle: () -> Unit) {
    Checkbox(
        checked = completed,
        onCheckedChange = { onToggle() },
        modifier = Modifier.semantics {
            contentDescription = "Task completion"
        }
    )
}
```

Focus: Semantic description for assistive tech

Exercise: Run Tests

In Android Studio or Terminal:

Exercise: Run Tests

In Android Studio or Terminal:

```
./gradlew test              # Unit tests
./gradlew connectedCheck    # UI tests (requires device)
```

Exercise: Run Tests

In Android Studio or Terminal:

```
./gradlew test              # Unit tests
./gradlew connectedCheck    # UI tests (requires device)
```

1. Run generated tests
2. Review test results
3. Use Composer to add one custom test
4. If test fails, use Chat: "Why is my test failing?"

Exercise: Run Tests

In Android Studio or Terminal:

```
./gradlew test              # Unit tests
./gradlew connectedCheck    # UI tests (requires device)
```

1. Run generated tests
2. Review test results
3. Use Composer to add one custom test
4. If test fails, use Chat: "Why is my test failing?"

Time: 5 minutes

Part 7: Advanced Topics

Production-Ready Polish

Hilt Dependency Injection

In Cursor Composer:

```
@codebase Add Hilt dependency injection to this project.
Create Application class annotated with @HiltAndroidApp,
create a module to provide TaskRepository and Room database,
and set up ViewModel injection. Add all necessary
dependencies to build.gradle.
```

Hilt Dependency Injection

In Cursor Composer:

```
@codebase Add Hilt dependency injection to this project.
Create Application class annotated with @HiltAndroidApp,
create a module to provide TaskRepository and Room database,
and set up ViewModel injection. Add all necessary
dependencies to build.gradle.
```

AI generates:

Hilt Dependency Injection

In Cursor Composer:

```
@codebase Add Hilt dependency injection to this project.
Create Application class annotated with @HiltAndroidApp,
create a module to provide TaskRepository and Room database,
and set up ViewModel injection. Add all necessary
dependencies to build.gradle.
```

AI generates:

- Application class
- Hilt modules
- ViewModel integration
- All annotations

Why Hilt?

Why Hilt?

- Compile-time DI

Why Hilt?

- Compile-time DI
- Less boilerplate

Why Hilt?

- Compile-time DI
- Less boilerplate
- Android-specific

Hilt Application Class

```
1  @HiltAndroidApp
2  class TaskApplication : Application()
```

Hilt Application Class

```
1   @HiltAndroidApp
2   class TaskApplication : Application()
```

Required setup:

Hilt Application Class

```
1   @HiltAndroidApp
2   class TaskApplication : Application()
```

Required setup:

- Register in `AndroidManifest.xml`
- Extend `Application()` class
- Single annotation: `@HiltAndroidApp`

Hilt Database Module

```
@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {
    @Provides
    @Singleton
    fun provideDatabase(@ApplicationContext context: Context): AppDatabase {
        return AppDatabase.getInstance(context)
    }
}
```

Hilt Repository Provider

```
@Module
@InstallIn(SingletonComponent::class)
object RepositoryModule {
    @Provides
    @Singleton
    fun provideTaskRepository(taskDao: TaskDao): TaskRepository {
        return TaskRepository(taskDao)
    }
}
```

Hilt ViewModel Injection

```
@HiltViewModel
class TaskViewModel @Inject constructor(
    private val repository: TaskRepository
) : ViewModel() {
    // ... ViewModel implementation
}

@Composable
fun TaskListScreen(
    viewModel: TaskViewModel = hiltViewModel() // Injected!
) {
    // ...
}
```

Key: `@HiltViewModel` + `@Inject` constructor + `hiltViewModel()`

Material 3 Theming

In Cursor Composer:

```
@codebase Create a Material 3 theme with custom color
scheme (purple primary) and add dark theme support using
dynamic colors. Apply this theme to the app.
```

Material 3 Theming

In Cursor Composer:

```
@codebase Create a Material 3 theme with custom color
scheme (purple primary) and add dark theme support using
dynamic colors. Apply this theme to the app.
```

AI generates:

Material 3 Theming

In Cursor Composer:

```
@codebase Create a Material 3 theme with custom color
scheme (purple primary) and add dark theme support using
dynamic colors. Apply this theme to the app.
```

AI generates:

- Color scheme (light + dark)
- Typography
- Shapes
- Theme composable

Material 3 Theming

In Cursor Composer:

```
@codebase Create a Material 3 theme with custom color
scheme (purple primary) and add dark theme support using
dynamic colors. Apply this theme to the app.
```

AI generates:

- Color scheme (light + dark)
- Typography
- Shapes
- Theme composable

Test dark mode in Android Studio!

Material 3 Color Schemes

```
private val LightColorScheme = lightColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

private val DarkColorScheme = darkColorScheme(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40
)
```

Material 3 Theme Composable

```
@Composable
fun TaskManagerTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S ->
            if (darkTheme) dynamicDarkColorScheme(LocalContext.current)
            else dynamicLightColorScheme(LocalContext.current)
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

    MaterialTheme(colorScheme = colorScheme, content = content)
}
```

Accessibility

Use Chat Mode for Review:

```
Chat: "Review TaskCard for accessibility. What improvements
would help screen reader users?"
```

Accessibility

Use Chat Mode for Review:

```
Chat: "Review TaskCard for accessibility. What improvements
would help screen reader users?"
```

Then use Composer to apply:

Accessibility

Use Chat Mode for Review:

```
Chat: "Review TaskCard for accessibility. What improvements
would help screen reader users?"
```

Then use Composer to apply:

```
@codebase Add semantic properties to TaskCard for better
accessibility
```

Accessibility

Use Chat Mode for Review:

```
Chat: "Review TaskCard for accessibility. What improvements
would help screen reader users?"
```

Then use Composer to apply:

```
@codebase Add semantic properties to TaskCard for better
accessibility
```

Key Properties:

Accessibility

Use Chat Mode for Review:

```
Chat: "Review TaskCard for accessibility. What improvements
would help screen reader users?"
```

Then use Composer to apply:

```
@codebase Add semantic properties to TaskCard for better
accessibility
```

Key Properties:

- `contentDescription` - Describe images/icons
- `Role.Checkbox` - Semantic roles
- `testTag` - For testing
- Minimum touch targets (48dp)

Performance Optimization

Use Chat Mode:

```
Chat: "Review TaskList for potential recomposition issues
and suggest optimizations"
```

Performance Optimization

Use Chat Mode:

```
Chat: "Review TaskList for potential recomposition issues
and suggest optimizations"
```

Common Optimizations:

Performance Optimization

Use Chat Mode:

```
Chat: "Review TaskList for potential recomposition issues
and suggest optimizations"
```

Common Optimizations:

- `remember` - Cache computations
- `derivedStateOf` - Calculated state
- `key` in LazyColumn - Proper item identification
- Immutable data classes
- Stable parameters

Performance Optimization

Use Chat Mode:

```
Chat: "Review TaskList for potential recomposition issues
and suggest optimizations"
```

Common Optimizations:

- `remember` - Cache computations
- `derivedStateOf` - Calculated state
- `key` in LazyColumn - Proper item identification
- Immutable data classes
- Stable parameters

Apply in Composer if needed

Exercise: Polish Your App

Using Composer and Chat:

Exercise: Polish Your App

Using Composer and Chat:

1. Add Hilt DI to your project
2. Apply Material 3 theme
3. Toggle dark mode in Android Studio
4. Use Chat to review accessibility
5. Apply accessibility improvements
6. Ask for performance review

Exercise: Polish Your App

Using Composer and Chat:

1. Add Hilt DI to your project
2. Apply Material 3 theme
3. Toggle dark mode in Android Studio
4. Use Chat to review accessibility
5. Apply accessibility improvements
6. Ask for performance review

Time: 5 minutes

Part 8: Now in Android

Learning from Google’s Sample

Now in Android Overview

Now in Android Overview

What is it?

Now in Android Overview

What is it?

- Official Google sample app
- Production-quality code
- Modern Android practices
- Fully open source

Now in Android Overview

What is it?

- Official Google sample app
- Production-quality code
- Modern Android practices
- Fully open source

What you'll learn:

Now in Android Overview

What is it?

- Official Google sample app
- Production-quality code
- Modern Android practices
- Fully open source

What you'll learn:

- Multi-module architecture
- Build conventions
- Advanced Compose patterns
- Comprehensive testing

Demo: Exploring with AI

Clone and open:

```
git clone https://github.com/android/nowinandroid
# Open in Android Studio (to run)
# Open in Cursor (to analyze)
```

Demo: Exploring with AI

Clone and open:

```
git clone https://github.com/android/nowinandroid
# Open in Android Studio (to run)
# Open in Cursor (to analyze)
```

In Cursor Chat Mode:

Demo: Exploring with AI

Clone and open:

```
git clone https://github.com/android/nowinandroid
# Open in Android Studio (to run)
# Open in Cursor (to analyze)
```

In Cursor Chat Mode:

```
Chat: "Explain the overall architecture of Now in Android app"
Chat: "What is the purpose of the :core:data module?"
Chat: "How is the navigation implemented?"
Chat: "What testing strategies are used?"
```

Demo: Exploring with AI

Clone and open:

```
git clone https://github.com/android/nowinandroid
# Open in Android Studio (to run)
# Open in Cursor (to analyze)
```

In Cursor Chat Mode:

```
Chat: "Explain the overall architecture of Now in Android app"
Chat: "What is the purpose of the :core:data module?"
Chat: "How is the navigation implemented?"
Chat: "What testing strategies are used?"
```

Run in Android Studio to see the real app

Multi-Module Architecture

```
nowinandroid/
├── app/                    # Main app module
├── core/
│   ├── data/              # Data layer
│   ├── database/          # Room database
│   ├── datastore/         # Preferences
│   ├── model/             # Data models
│   ├── network/           # API calls
│   └── ui/                # Common UI
├── feature/
│   ├── foryou/            # For You screen
│   ├── interests/         # Interests screen
│   └── bookmarks/         # Bookmarks screen
└── build-logic/           # Build configuration
```

Multi-Module Architecture

```
nowinandroid/
├── app/                    # Main app module
├── core/
│   ├── data/              # Data layer
│   ├── database/          # Room database
│   ├── datastore/         # Preferences
│   ├── model/             # Data models
│   ├── network/           # API calls
│   └── ui/                # Common UI
├── feature/
│   ├── foryou/            # For You screen
│   ├── interests/         # Interests screen
│   └── bookmarks/         # Bookmarks screen
└── build-logic/           # Build configuration
```

Benefits:

Multi-Module Architecture

```
nowinandroid/
├── app/                    # Main app module
├── core/
│   ├── data/              # Data layer
│   ├── database/          # Room database
│   ├── datastore/         # Preferences
│   ├── model/             # Data models
│   ├── network/           # API calls
│   └── ui/                # Common UI
├── feature/
│   ├── foryou/            # For You screen
│   ├── interests/         # Interests screen
│   └── bookmarks/         # Bookmarks screen
└── build-logic/           # Build configuration
```

Benefits:

- Clear separation of concerns
- Parallel builds
- Reusable modules
- Team scalability

Exercise: Explore Now in Android

In Cursor Chat Mode, ask:

Exercise: Explore Now in Android

In Cursor Chat Mode, ask:

1. "How many feature modules are there?"
2. "Explain the DI setup in this app"
3. "What design system components are used?"
4. "How is offline support implemented?"
5. "What testing patterns are used?"

Exercise: Explore Now in Android

In Cursor Chat Mode, ask:

1. "How many feature modules are there?"
2. "Explain the DI setup in this app"
3. "What design system components are used?"
4. "How is offline support implemented?"
5. "What testing patterns are used?"

Compare with your Task Manager app

Exercise: Explore Now in Android

In Cursor Chat Mode, ask:

1. "How many feature modules are there?"
2. "Explain the DI setup in this app"
3. "What design system components are used?"
4. "How is offline support implemented?"
5. "What testing patterns are used?"

Compare with your Task Manager app

Time: 5 minutes

Wrap-Up: Key Takeaways

What We Built Today

What We Built Today

✅ Complete Android App

What We Built Today

✅ Complete Android App

- Jetpack Compose UI with Material 3
- ViewModels with StateFlow
- Multi-screen navigation
- Room database persistence
- Hilt dependency injection
- Comprehensive test suite
- Dark theme support
- Accessibility features

What We Built Today

✅ Complete Android App

- Jetpack Compose UI with Material 3
- ViewModels with StateFlow
- Multi-screen navigation
- Room database persistence
- Hilt dependency injection
- Comprehensive test suite
- Dark theme support
- Accessibility features

All with AI assistance!

The Hybrid Workflow Decision Tree

```
Need to understand code/APIs?
  → Use Chat mode (Cmd/Ctrl+L)

Need to generate/modify code?
  → Use Composer (Cmd/Ctrl+Shift+I) with @codebase

Need to run/test app?
  → Switch to Android Studio

Need to debug layout?
  → Use Chat to explain, then Composer to fix

Need to add feature?
  → Use Composer with detailed instructions + @codebase

Need multi-file changes?
  → Always use Composer, never manual editing
```

Best Practices We Learned

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results
3. **Use each tool for its strength** - Cursor for code, Android Studio for running

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results
3. **Use each tool for its strength** - Cursor for code, Android Studio for running
4. **Iterate with AI** - Start simple, refine with follow-ups

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results
3. **Use each tool for its strength** - Cursor for code, Android Studio for running
4. **Iterate with AI** - Start simple, refine with follow-ups
5. **Review AI output** - Don’t blindly accept

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results
3. **Use each tool for its strength** - Cursor for code, Android Studio for running
4. **Iterate with AI** - Start simple, refine with follow-ups
5. **Review AI output** - Don’t blindly accept
6. **Test frequently** - Run app after each feature

Best Practices We Learned

1. **Always tag** `@codebase` - Give AI full context
2. **Be specific in prompts** - More detail = better results
3. **Use each tool for its strength** - Cursor for code, Android Studio for running
4. **Iterate with AI** - Start simple, refine with follow-ups
5. **Review AI output** - Don’t blindly accept
6. **Test frequently** - Run app after each feature
7. **Ask Chat before generating** - Understand concepts first

Common Pitfalls to Avoid — Don’t

Common Pitfalls to Avoid — Don’t

❌ **Don’t:**

Common Pitfalls to Avoid — Don’t

❌ **Don’t:**

- Use Cursor exclusively (need Android Studio too)
- Forget `@codebase` tag
- Accept AI code without review
- Skip testing
- Ignore accessibility
- Over-rely on AI for debugging

Common Pitfalls to Avoid — Do

Common Pitfalls to Avoid — Do

✅ **Do:**

Common Pitfalls to Avoid — Do

✅ **Do:**

- Use hybrid workflow
- Write detailed prompts
- Review and understand generated code
- Test on real devices
- Ask Chat mode for explanations
- Use AI as a learning tool

Resources

Official Documentation:

- Jetpack Compose
- Android Architecture Guide
- Room Database
- Hilt DI
- Now in Android

Learning:

- Compose Pathway
- Testing in Compose
- Kotlin Coroutines

Questions?

Open Q&A

Ask about anything we covered today:

- Hybrid workflow
- Composer mode
- Jetpack Compose
- State management
- Room database
- Testing
- Your own projects

Thank You!

Contact:

- ken.kousen@kousenit.com
- http://kousenit.com
- @kenkousen

Thanks for joining!

- Slides and code are in the GitHub repo
- Reach out with questions anytime

Happy Coding with AI! 🚀