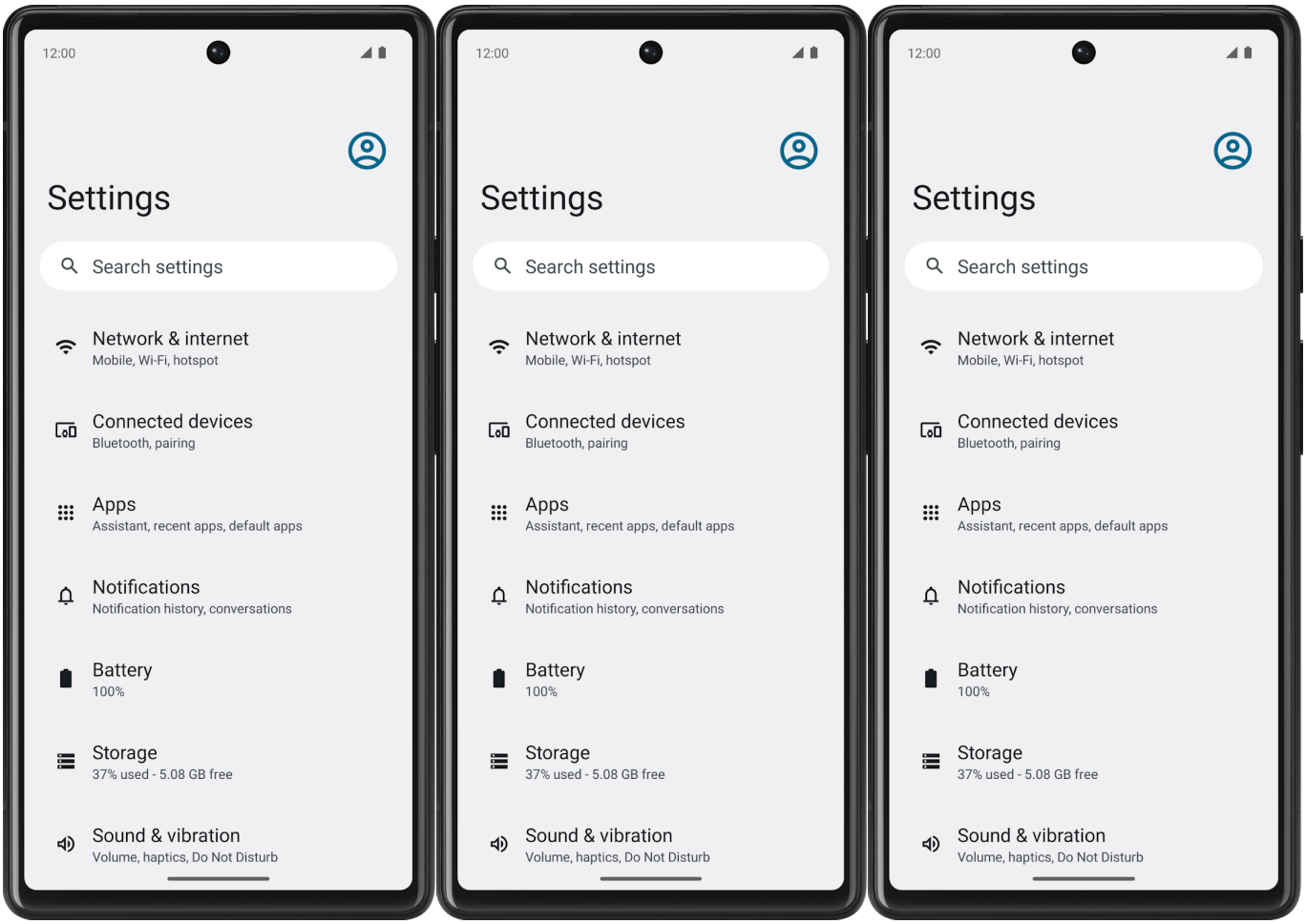
Навигация в Jetpack Compose

Навигация между экранами с помощью Compose

Прежде чем начать

До этого момента приложения, над которыми вы работали, состояли из одного экрана. Однако во многих приложениях, которыми вы пользуетесь, наверняка есть несколько экранов, по которым можно перемещаться. Например, приложение «Настройки» содержит множество страниц, разбросанных по разным экранам.



В современной Android-разработке многоэкранные приложения создаются с помощью компонента `Jetpack Navigation`. Компонент Navigation Compose позволяет легко создавать многоэкранные приложения в Compose, используя декларативный подход, подобно созданию пользовательских интерфейсов. В этой работе вы узнаете об основных возможностях компонента Navigation Compose, о том, как сделать `AppBar` отзывчивым, как отправлять данные из вашего приложения в другое приложение с помощью намерений - и все это с демонстрацией лучших практик работы со все более сложными приложениями.

Предварительные условия

- Знакомство с языком Kotlin, включая типы функций, лямбды и функции области видимости

- Знакомство с основными макетами строк и столбцов в Compose

Что вы узнаете

- Создавать композитный `NavHost` для определения маршрутов и экранов в вашем приложении.
- Перемещаться между экранами с помощью `NavHostController`.
- Манипулировать задним стеком для перехода к предыдущим экранам.
- Используйте намерения для обмена данными с другим приложением.
- Настройте `AppBar`, включая заголовок и кнопку «Назад».

Что вы создадите

- Вы реализуете навигацию в многоэкранном приложении.

Что вам понадобится

- Последняя версия Android Studio
- Подключение к серверу для загрузки начального кода

Скачайте стартовый код

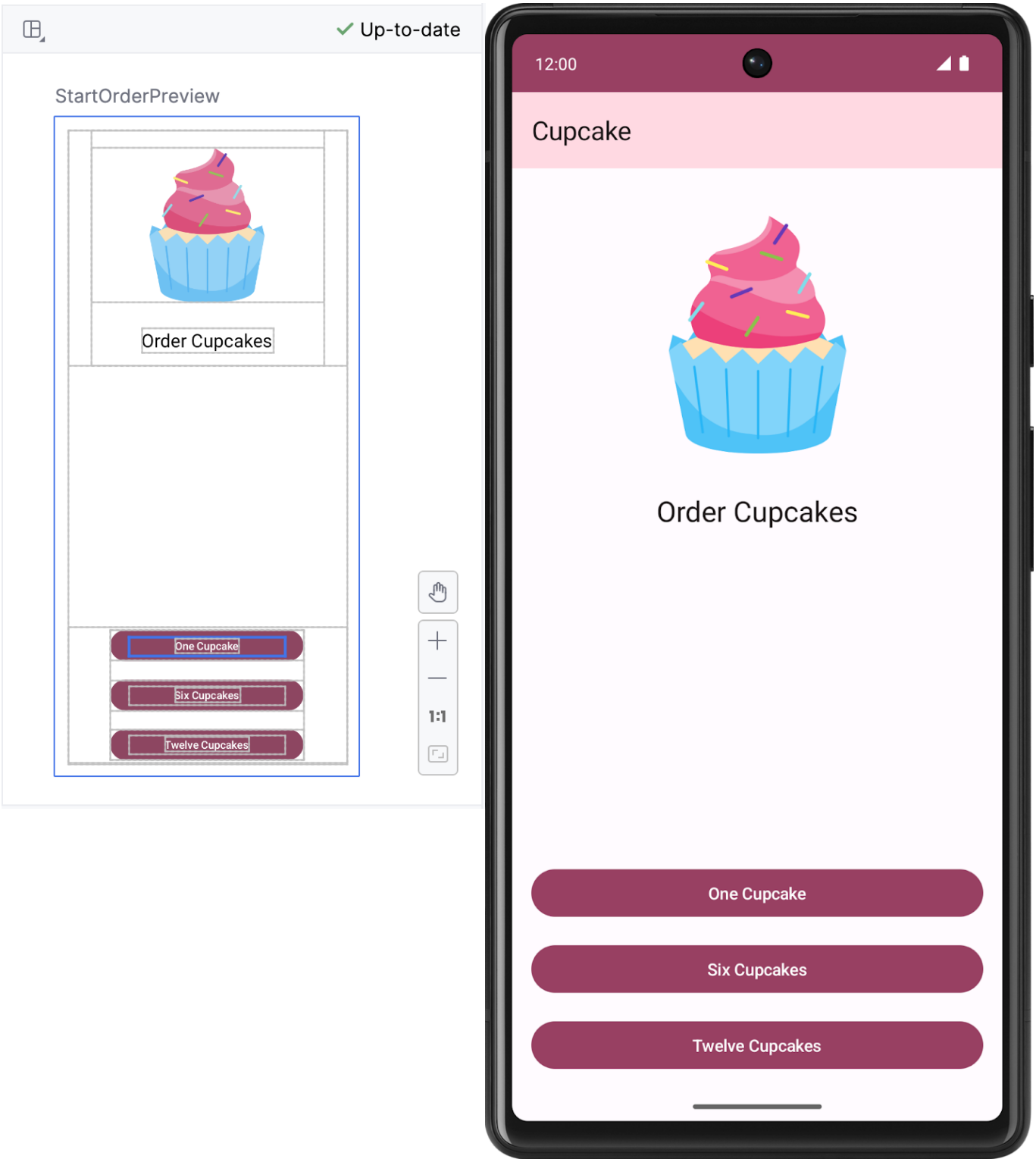
- скачайте в repositories репозиторий capcake
- распакуйте и перейдите в проект
- git checkout starter

Знакомство с приложением

Приложение `Cupcake` немного отличается от приложений, с которыми вы работали до сих пор. Вместо того чтобы отображать весь контент на одном экране, приложение имеет четыре отдельных экрана, и пользователь может перемещаться по каждому из них, заказывая кексы. Если вы запустите приложение, то ничего не увидите и не сможете перемещаться между этими экранами, поскольку компонент навигации еще не добавлен в код приложения. Тем не менее, вы можете посмотреть композитные превью для каждого экрана и сопоставить их с финальными экранами приложения, представленными ниже.

Стартовый экран заказа

На первом экране пользователю предлагаются три кнопки, соответствующие количеству кексов, которые необходимо заказать.

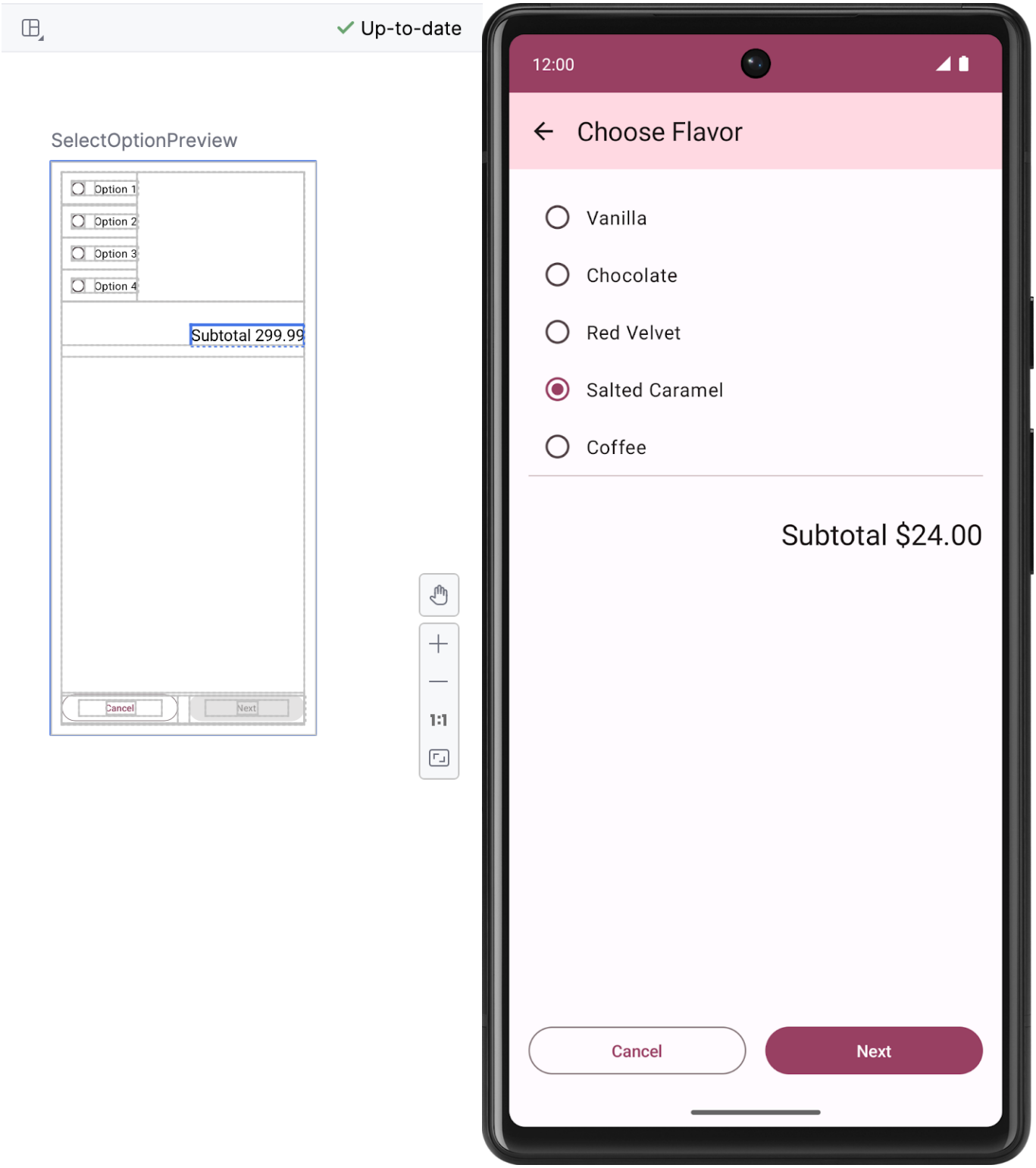


В коде это представлено композитным экраном `StartOrderScreen` в файле `StartOrderScreen.kt`.

Экран состоит из одной колонки с изображением и текстом, а также трех пользовательских кнопок для заказа различных количеств кексов. Пользовательские кнопки реализованы с помощью композита `SelectQuantityButton`, который также находится в `StartOrderScreen.kt`.

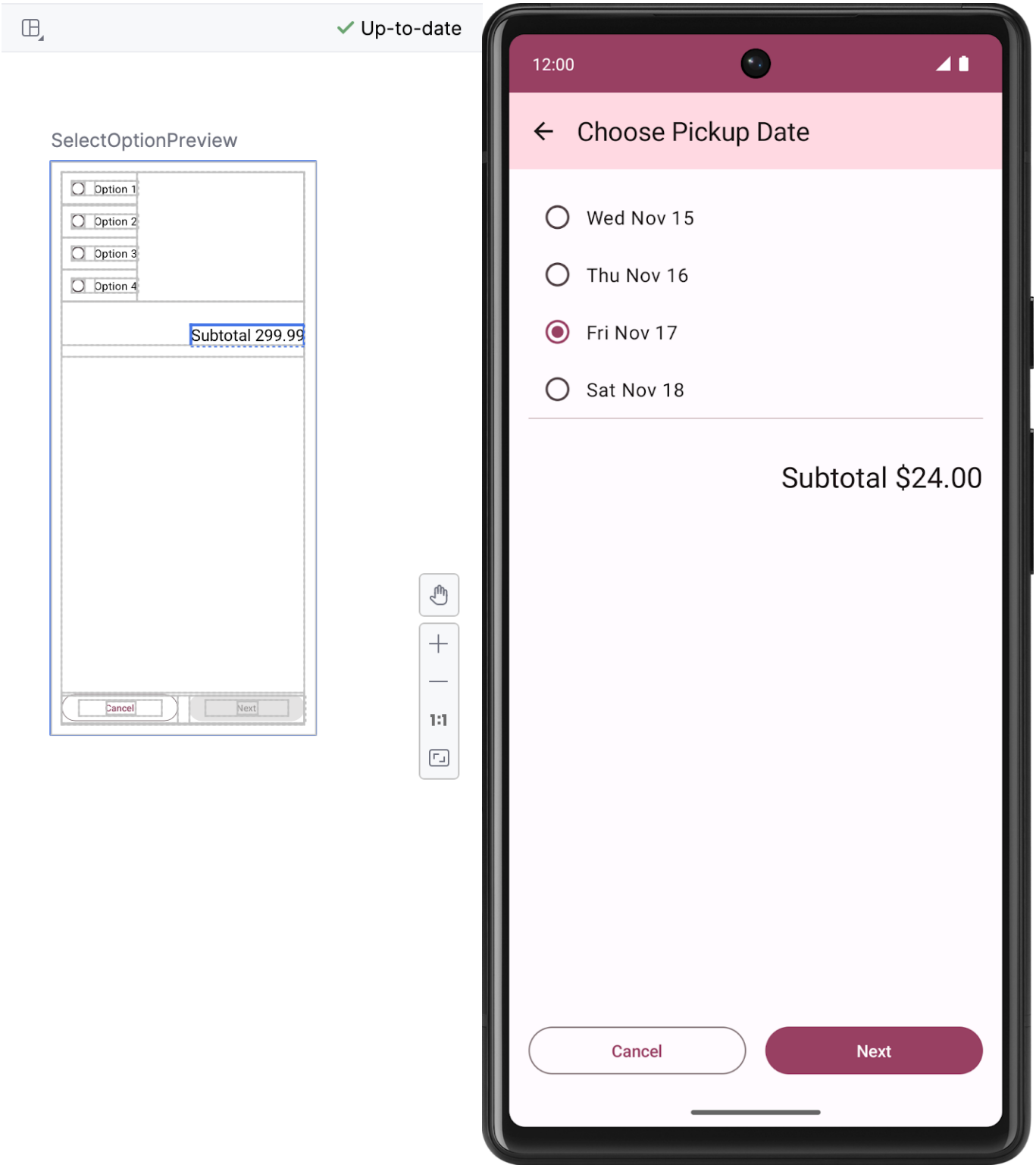
Экран выбора вкуса

После выбора количества приложение предлагает пользователю выбрать вкус кекса. Приложение использует так называемые радиокнопки для отображения различных вариантов. Пользователь может выбрать один вкус из нескольких возможных.



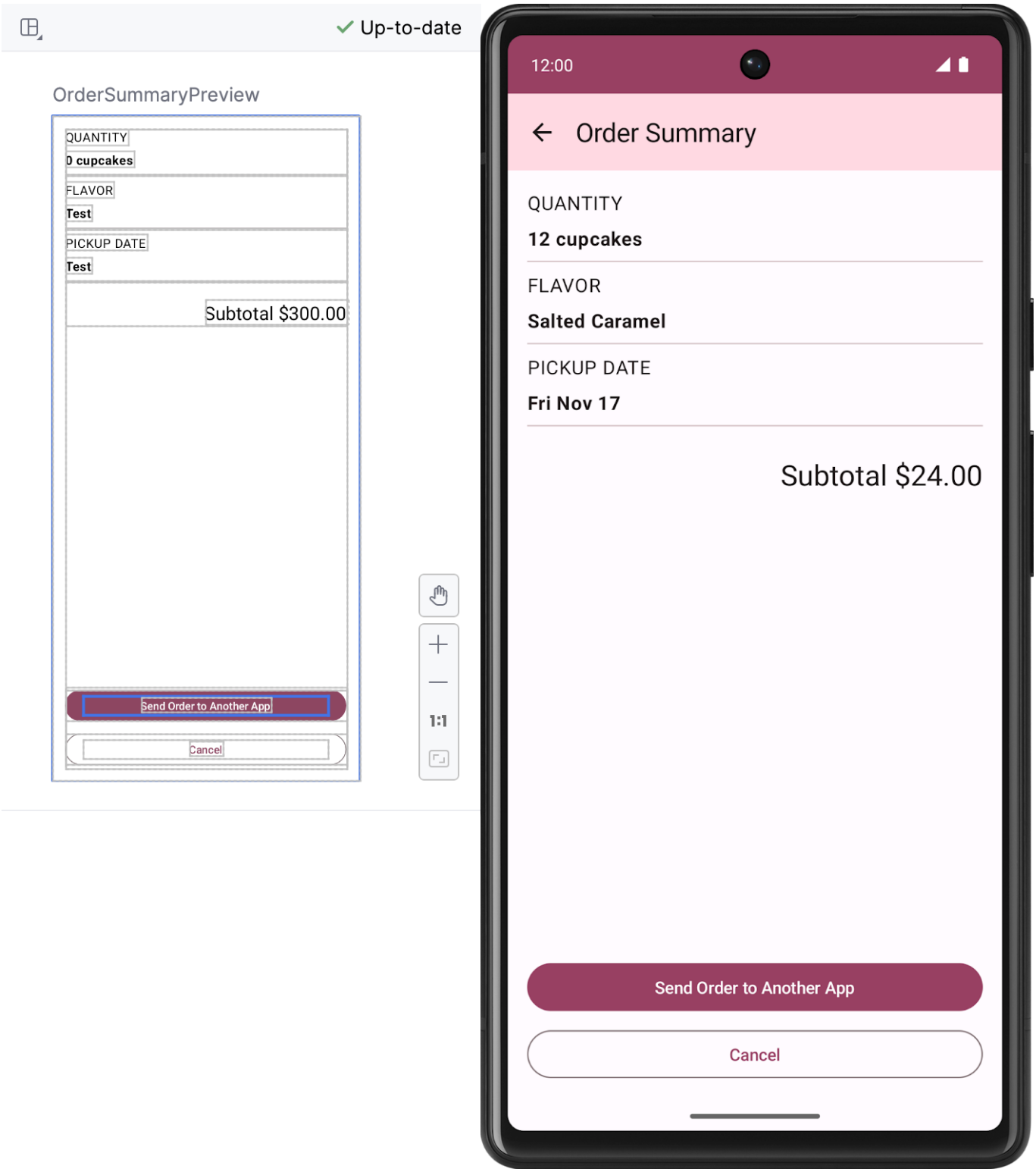
Список возможных вкусов хранится в виде списка строковых идентификаторов ресурсов в `data.DataSource.kt`.

Экран выбора даты самовывоза После выбора аромата приложение представляет пользователям еще одну серию радиокнопок для выбора даты самовывоза. Варианты забора берутся из списка, возвращаемого функцией `pickupOptions()` в `OrderViewModel`.



И экран выбора вкуса, и экран выбора даты забора представлены одним и тем же композитом `SelectOptionScreen` в файле `SelectOptionScreen.kt`. Почему используется один и тот же компонент? Макет этих экранов абсолютно одинаков! Единственное различие - это данные, но вы можете использовать один и тот же компонент для отображения экранов вкуса и даты сбора.

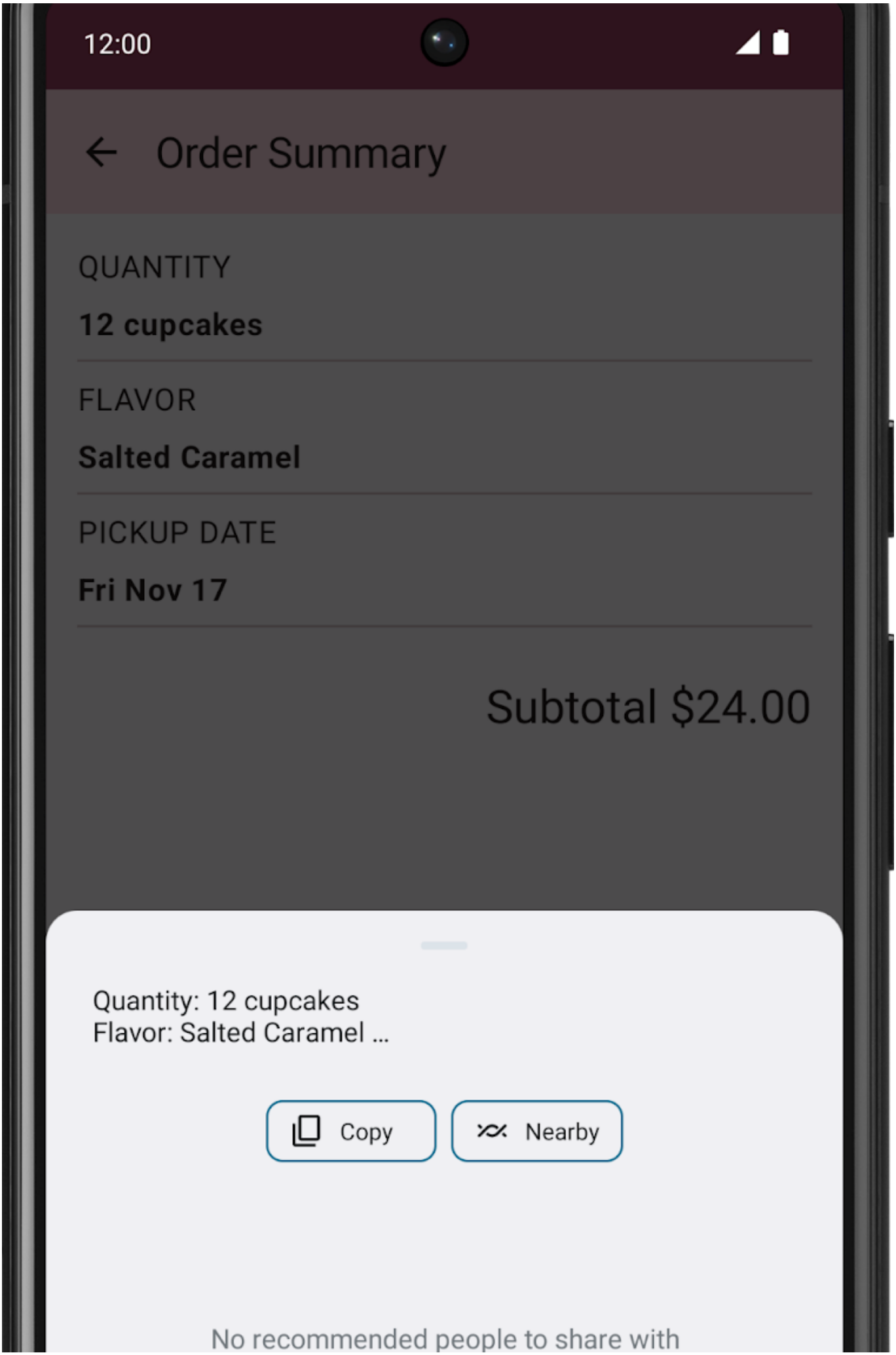
Экран сводки заказа После выбора даты получения заказа приложение отображает экран «Сводка заказов», где пользователь может просмотреть и завершить заказ.

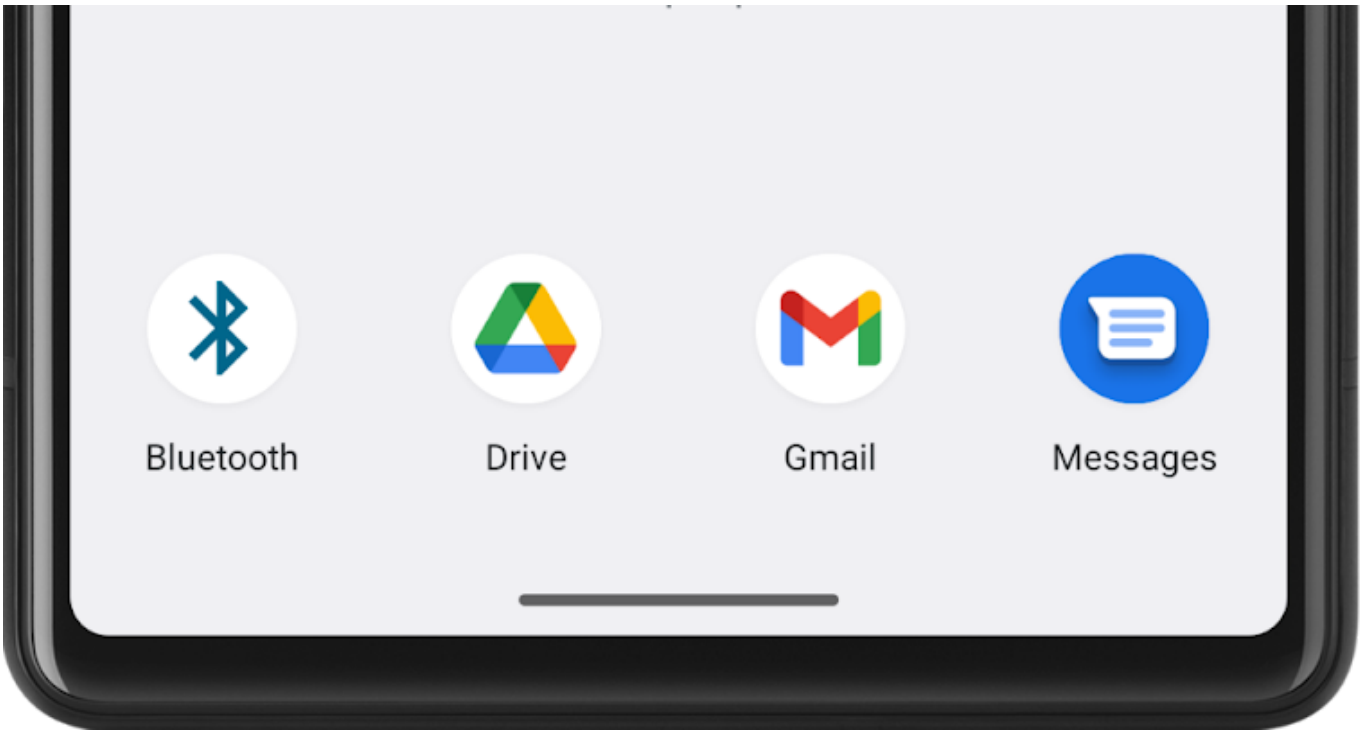


Этот экран реализуется с помощью экрана `OrderSummaryScreen`, создаваемого в файле SummaryScreen.kt.

Макет состоит из колонки, содержащей всю информацию о заказе, текста, который можно использовать для подведения итогов, и кнопок для отправки заказа в другое приложение или отмены заказа и возврата к первому экрану.

Если пользователь решил отправить заказ в другое приложение, приложение `Cupcake` отображает `Android ShareSheet`, в котором показаны различные варианты отправки.


12:00
Order Summary
QUANTITY
12 cupcakes
FLAVOR
Salted Caramel
PICKUP DATE
Fri Nov 17
Subtotal $24.00
Quantity: 12 cupcakes
Flavor: Salted Caramel ...
Copy
Nearby
No recommended people to share with



{style="width:400px"}

Текущее состояние приложения хранится в `data.OrderUiState.kt`. Класс данных `OrderUiState` содержит свойства для хранения выбора пользователя на каждом экране.

Экраны приложения будут представлены в композите `CupcakeApp`. Однако в стартовом проекте приложение просто показывает первый экран. В настоящее время нет возможности перемещаться по всем экранам приложения, но не волнуйтесь, вы здесь именно для этого! Вы узнаете, как определить маршруты навигации, настроить композит `NavHost` для перехода между экранами - также известные как пункты назначения - выполнить намерения для интеграции с системными компонентами пользовательского интерфейса, такими как экран обмена, и заставить `AppBar` реагировать на изменения навигации.

Многократно используемые составные элементы Там, где это уместно, примеры приложений в этом курсе разработаны для реализации лучших практик. Приложение `Cupcake` не является исключением. В пакете `ui.components` вы увидите файл `CommonUi.kt`, который содержит составную метку `FormattedPriceLabel`. Несколько экранов в приложении используют этот компонент для последовательного форматирования цены заказа. Вместо того чтобы дублировать один и тот же текст с одним и тем же форматированием и модификаторами, вы можете определить `FormattedPriceLabel` один раз, а затем повторно использовать его столько раз, сколько потребуется для других экранов.

Экраны вкуса и даты получения используют составной экран `SelectOptionScreen`, который также является многоразовым. Этот составной экран принимает параметр options типа `List<String>`, который представляет опции для отображения. Опции отображаются в строке, состоящей из композита `RadioButton` и композита `Text`, содержащего каждую строку. Столбец окружает весь макет и также содержит композит `Text` для отображения форматированной цены, кнопку `Cancel` и кнопку `Next`.

Определение маршрутов и создание NavHostController

Части компонента Navigation

Компонент `Navigation` состоит из трех основных частей:

- **NavController**: Отвечает за навигацию между пунктами назначения, то есть экранами в вашем приложении.
- **NavGraph**: Отображает составные пункты назначения для навигации.
- **NavHost**: Составной элемент, выступающий в качестве контейнера для отображения текущего пункта назначения в NavGraph.

В уроке вы сосредоточитесь на `NavController` и `NavHost`. Внутри `NavHost` вы определите пункты назначения для `NavGraph` приложения `Cupcake`.

Определение маршрутов для пунктов назначения в вашем приложении

Одной из фундаментальных концепций навигации в приложении Compose является **маршрут**. Маршрут - это строка, которая соответствует пункту назначения. Эта идея схожа с концепцией URL. Точно так же, как разные URL-адреса указывают на разные страницы веб-сайта, маршрут - это строка, которая указывает на место назначения и служит его уникальным идентификатором. **Пункт назначения** - это, как правило, один компонент или группа компонентов, соответствующих тому, что видит пользователь. В приложении `Cupcake` нужны пункты назначения для экрана начала заказа, экрана вкуса, экрана даты получения и экрана сводки заказа.

В приложении существует конечное число экранов, поэтому существует и конечное число маршрутов. Вы можете определить маршруты приложения с помощью класса **enum**.

> Классы enum в Kotlin имеют свойство name, которое возвращает строку с именем свойства.

Начнем с определения четырех маршрутов приложения `Cupcake`.

- Старт: Выберите количество кексов с помощью одной из трех кнопок.
- Вкус: Выберите вкус из списка вариантов.
- Забрать: Выберите дату забора из списка вариантов.
- Подведение итогов: Просмотр выбранных вариантов и отправка или отмена заказа.

Добавьте класс enum для определения маршрутов.

- В файле `CupcakeScreen.kt`, над композитом `CupcakeAppBar`, добавьте класс перечисления `CupcakeScreen`.

```
enum class CupcakeScreen() {

}
```

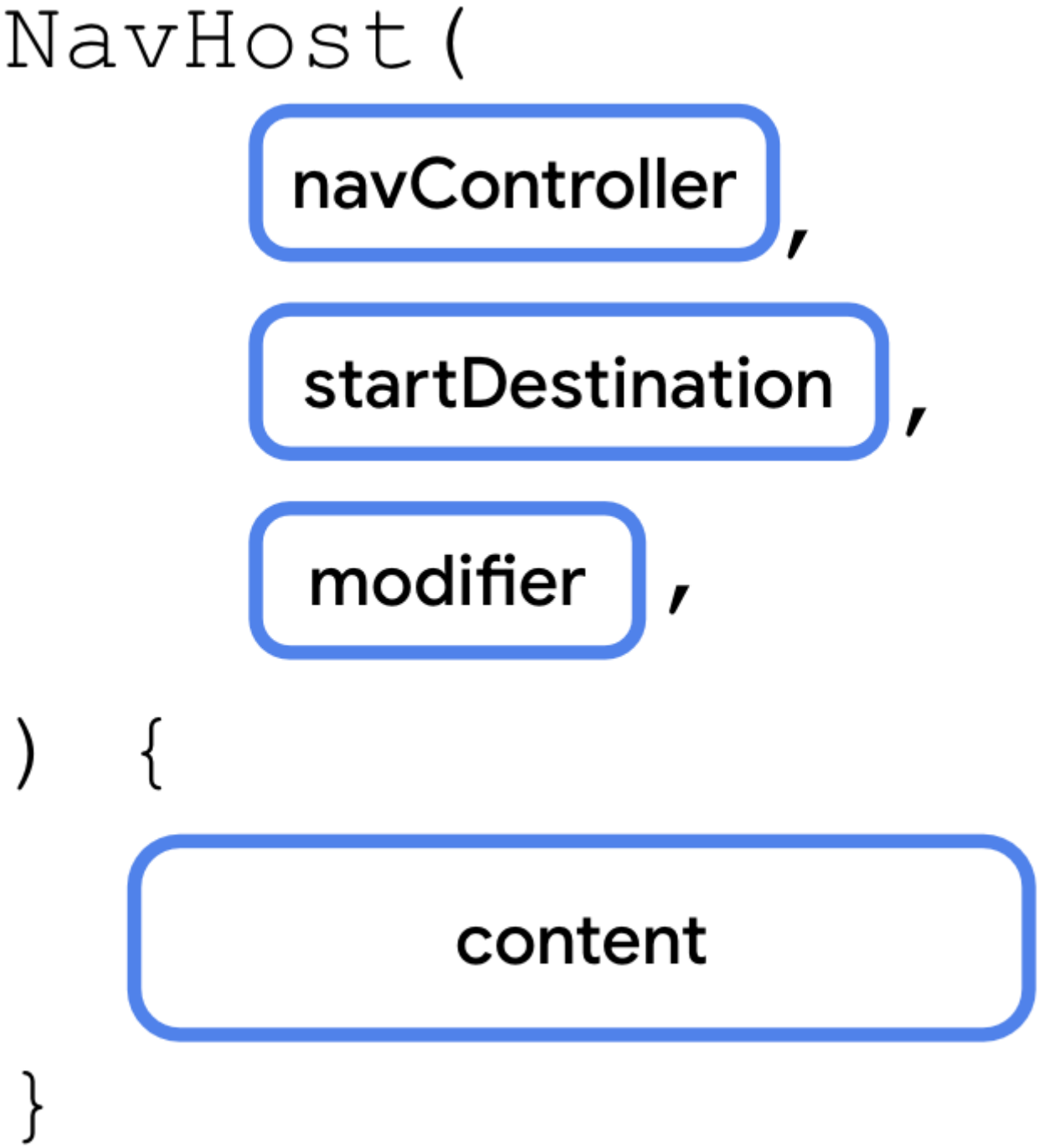
- Добавьте четыре случая в класс перечисления: Start, Flavor, Pickup и Summary.

```
enum class CupcakeScreen() {
    Start,
    Flavor,
    Pickup,
    Summary
}
```

- Добавьте `NavHost` в свое приложение

NavHost - это компонент, который отображает другие компонентные пункты назначения, основанные на заданном маршруте. Например, если маршрут - Flavor, NavHost отобразит экран для выбора вкуса кекса. Если маршрут - Summary, то приложение отобразит экран сводки.

Синтаксис для NavHost такой же, как и для любого другого Composable.



Есть два важных параметра.

- **navController**: Экземпляр класса `NavHostController`. Вы можете использовать этот объект для навигации между экранами, например, вызывая метод `navigate()` для перехода к другому

пункту назначения. Вы можете получить `NavHostController`, вызвав `rememberNavController()` из композитной функции.

- **startDestination**: Строковый маршрут, определяющий пункт назначения, показываемый по умолчанию, когда приложение впервые отображает `NavHost`. В случае с приложением `Cupcake` это должен быть маршрут Start. Как и другие композиты, NavHost также принимает параметр-модификатор.

> Примечание: NavHostController - это подкласс класса NavController, который предоставляет дополнительную функциональность для использования с композитным NavHost.

Вы добавите NavHost в составной `CupcakeApp` в файле `CupcakeScreen.kt`. Во-первых, вам нужна ссылка на контроллер навигации. Контроллер навигации можно использовать как в NavHost, который вы добавляете сейчас, так и в `AppBar`, который вы добавите на следующем этапе. Поэтому вам следует объявить переменную в композите `CupcakeApp()`.

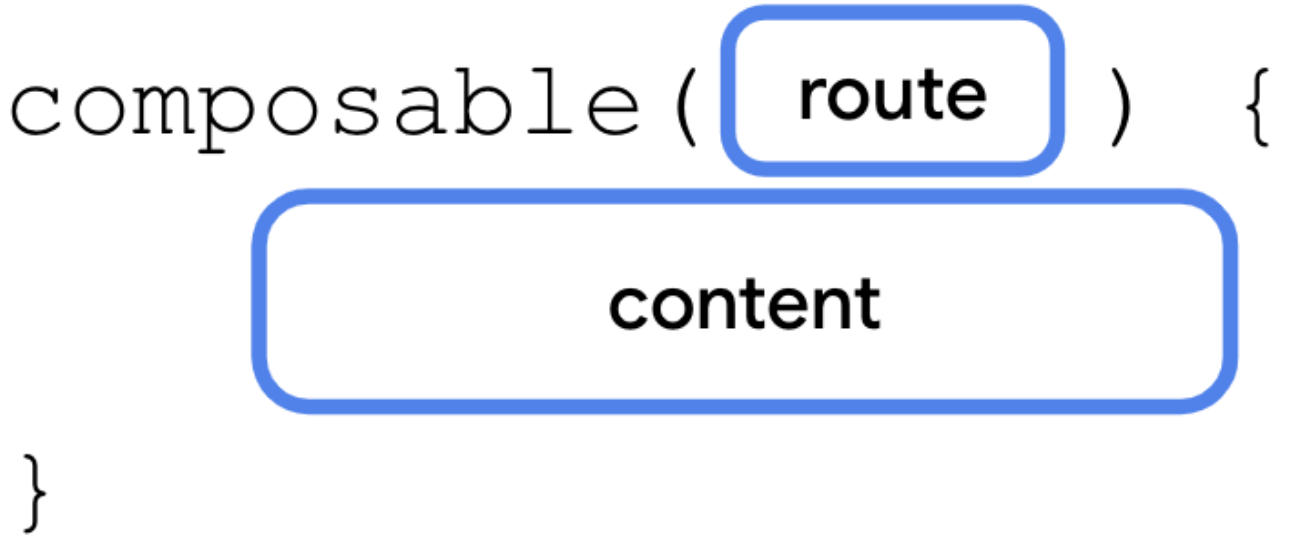
- Откройте файл `CupcakeScreen.kt`. Внутри `Scaffold`, под переменной `uiState`, добавьте составной NavHost.

```
import androidx.navigation.compose.NavHost

Scaffold(
    ...
) { innerPadding ->
    val uiState by viewModel.uiState.collectAsState()

    NavHost()
}
```

- Передайте переменную `navController` для параметра navController и `CupcakeScreen.Start.name` для параметра startDestination. Передайте модификатор, который был передан в `CupcakeApp()`, для параметра modifier. В качестве последнего параметра передайте пустую лямбду в конце.

```
import androidx.compose.foundation.layout.padding

NavHost(
    navController = navController,
    startDestination = CupcakeScreen.Start.name,
    modifier = Modifier.padding(innerPadding)
) {

}
```

Работа с маршрутами в NavHost Как и другие составные элементы, NavHost принимает тип функции для своего содержимого.



В функции содержимого NavHost вызывается функция composable(). Функция composable() имеет два обязательных параметра.

- **route**: Строка, соответствующая имени маршрута. Это может быть любая уникальная строка. Вы будете использовать свойство name из констант перечисления `CupcakeScreen`.
- **content** (содержимое): Здесь вы можете вызвать композит, который вы хотите отобразить для данного маршрута. Вы будете вызывать функцию composable() один раз для каждого из четырех маршрутов.

> Примечание: Функция composable() является функцией расширения `NavGraphBuilder`.

- Вызовите функцию composable(), передав в качестве маршрута `CupcakeScreen.Start.name`.

```
import androidx.navigation.compose.composable

NavHost(
    navController = navController,
    startDestination = CupcakeScreen.Start.name,
    modifier = Modifier.padding(innerPadding)
) {
    composable(route = CupcakeScreen.Start.name) {

    }
}
```

- В последующей лямбде вызовите композит `StartOrderScreen`, передав свойство `quantityOptions` для свойства `quantityOptions`. Для модификатора передайте Modifier.fillMaxSize().padding(dimensionResource(R.dimen.padding_medium))

```
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.ui.res.dimensionResource
import com.example.cupcake.ui.StartOrderScreen
import com.example.cupcake.data.DataSource
```

```
NavHost(
    navController = navController,
    startDestination = CupcakeScreen.Start.name,
    modifier = Modifier.padding(innerPadding)
) {
    composable(route = CupcakeScreen.Start.name) {
        StartOrderScreen(
            quantityOptions = DataSource.quantityOptions,
            modifier = Modifier
                .fillMaxSize()
                .padding(dimensionResource(R.dimen.padding_medium))
        )
    }
}
```

> Примечание: свойство `quantityOptions` берется из объекта-синглтона `DataSource` в `DataSource.kt`.

- После первого вызова composable() снова вызовите composable(), передав в качестве маршрута `CupcakeScreen.Flavor.name`.

```
composable(route = CupcakeScreen.Flavor.name) {

}
```

Внутри последующей лямбды получите ссылку на `LocalContext.current` и сохраните ее в переменной с именем context. **Context** - это абстрактный класс, реализация которого обеспечивается системой Android. Он позволяет получать доступ к ресурсам и классам, специфичным для приложения, а также вызывать операции на уровне приложения, такие как запуск действий и т. д. Вы можете использовать эту переменную для получения строк из списка идентификаторов ресурсов в модели представления для отображения списка вкусов.

```
import androidx.compose.ui.platform.LocalContext

composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
}
```

- Вызовите композитный экран `SelectOptionScreen`.

```
composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(


```

```
    )
}
```

- Экран вкуса должен отображать и обновлять промежуточный итог, когда пользователь выбирает вкус. Передайте `uiState.price` для параметра `subtotal`.

```
composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(
        subtotal = uiState.price
    )
}
```

- Экран вкусов получает список вкусов из строковых ресурсов приложения. Преобразуйте список идентификаторов ресурсов в список строк с помощью функции `map()` и вызова `context.resources.getString(id)` для каждого аромата.

```
import com.example.cupcake.ui.SelectOptionScreen

composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(
        subtotal = uiState.price,
        options = DataSource.flavors.map { id -> context.resources.getString(id) }
    )
}
```

Для параметра `onSelectionChanged` передайте лямбда-выражение, которое вызывает `setFlavor()` на модели представления, передавая в него (аргумент, переданный в `onSelectionChanged()`). Для параметра модификатора передайте `Modifier.fillMaxHeight()`.

```
import androidx.compose.foundation.layout.fillMaxHeight
import com.example.cupcake.data.DataSource.flavors

composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(
        subtotal = uiState.price,
        options = DataSource.flavors.map { id -> context.resources.getString(id)
},
        onSelectionChanged = { viewModel.setFlavor(it) },
        modifier = Modifier.fillMaxHeight()
    )
}
```

Экран «Дата сбора» аналогичен экрану «Вкус». Единственное отличие - это данные, передаваемые в составную функцию `SelectOptionScreen`.

- Снова вызовите функцию composable(), передав в качестве параметра `route` параметр `CupcakeScreen.Pickup.name`.

```
composable(route = CupcakeScreen.Pickup.name) {

}
```

- В последующей лямбде вызовите составной `SelectOptionScreen` и передайте `uiState.price` для параметра subtotal, как и раньше. Передайте `uiState.pickupOptions` для параметра `options` и лямбда-выражение, вызывающее `setDate()` на viewModel для параметра `onSelectionChanged`. Для параметра модификатора передайте `Modifier.fillMaxHeight()`.

```
SelectOptionScreen(
    subtotal = uiState.price,
    options = uiState.pickupOptions,
    onSelectionChanged = { viewModel.setDate(it) },
    modifier = Modifier.fillMaxHeight()
)
```

- Вызовите composable() еще раз, передав `CupcakeScreen.Summary.name` для маршрута.

```
composable(route = CupcakeScreen.Summary.name) {

}
```

- В последующей лямбде вызовите композит `OrderSummaryScreen()`, передав переменную `uiState` для параметра `orderUiState`. Для параметра модификатора передайте `Modifier.fillMaxHeight()`.

```
import com.example.cupcake.ui.OrderSummaryScreen

composable(route = CupcakeScreen.Summary.name) {
    OrderSummaryScreen(
        orderUiState = uiState,
        modifier = Modifier.fillMaxHeight()
    )
}
```

На этом настройка `NavHost` завершена. В следующем разделе вы заставите свое приложение менять маршруты и перемещаться между экранами при нажатии пользователем каждой из кнопок.

Навигация между маршрутами

Теперь, когда вы определили маршруты и привязали их к композитам в `NavHost`, пришло время перемещаться между экранами. За навигацию между маршрутами отвечает свойство `NavHostController` - navController, получаемое при вызове `rememberNavController()` -. Заметьте, однако, что это свойство определено в композите `CupcakeApp`. Вам нужен способ доступа к нему с разных экранов вашего приложения.

Легко, правда? Просто передайте `navController` в качестве параметра каждому из компонент.

Хотя этот подход работает, он не является идеальным способом создания архитектуры приложения. Преимущество использования `NavHost` для управления навигацией в приложении заключается в том, что логика навигации отделена от пользовательского интерфейса. Этот вариант позволяет избежать некоторых из основных недостатков передачи `navController` в качестве параметра.

Логика навигации хранится в одном месте, что может упростить сопровождение кода и предотвратить ошибки, не давая случайно отдельным экранам свободно управлять навигацией в вашем приложении. В приложениях, которые должны работать в разных форм-факторах (например, телефон в портретном режиме, складной телефон или планшет с большим экраном), кнопка может вызывать или не вызывать навигацию в зависимости от макета приложения. Отдельные экраны должны быть самодостаточными и не должны знать о других экранах в приложении. Вместо этого мы передаем в каждый composable тип функции для того, что должно произойти, когда пользователь нажмет на кнопку. Таким образом, композит и все его дочерние композиты сами решают, когда вызывать функцию. Однако логика навигации не передается на отдельные экраны вашего приложения. Все поведение навигации обрабатывается в `NavHost`.

- Добавьте обработчики кнопок в `StartOrderScreen` Начнем с добавления параметра типа function, который вызывается при нажатии одной из кнопок количества на первом экране. Эта функция передается в составной элемент `StartOrderScreen` и отвечает за обновление вью-модели и переход к следующему экрану.
- Откройте файл `StartOrderScreen.kt`. Ниже параметра `quantityOptions` и перед параметром modifier добавьте параметр `onNextButtonClicked` типа `() -> Unit`.

```
@Composable
fun StartOrderScreen(
    quantityOptions: List<Pair<Int, Int>>,
    onNextButtonClicked: () -> Unit,
    modifier: Modifier = Modifier
){
    ...
}
```

- Теперь, когда композит `StartOrderScreen` ожидает значение для `onNextButtonClicked`, найдите `StartOrderPreview` и передайте пустое лямбда-тело параметру `onNextButtonClicked`.

```
@Preview
@Composable
```

```
fun StartOrderPreview() {
    CupcakeTheme {
        StartOrderScreen(
            quantityOptions = DataSource.quantityOptions,
            onNextButtonClicked = {},
            modifier = Modifier
                .fillMaxSize()
                .padding(dimensionResource(R.dimen.padding_medium))
        )
    }
}
```

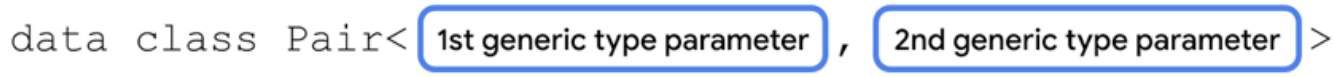
Каждая кнопка соответствует различному количеству кексов. Эта информация понадобится вам для того, чтобы функция, переданная для `onNextButtonClicked`, могла соответствующим образом обновить вью-модель.

- Измените тип параметра `onNextButtonClicked` так, чтобы он принимал параметр Int.

```
onNextButtonClicked: (Int) -> Unit,
```

- Чтобы получить Int для передачи при вызове `onNextButtonClicked()`, посмотрите на тип параметра `quantityOptions`.

Тип - `List<Pair<Int, Int>>` или список `Pair<Int, Int>`. Тип `Pair` может быть вам незнаком, но это, как следует из названия, пара значений. `Pair` принимает два параметра общего типа. В данном случае они оба имеют тип Int.



Доступ к каждому элементу в паре осуществляется либо через первое, либо через второе свойство. В случае с параметром `quantityOptions` композитного экрана `StartOrderScreen` первый Int - это идентификатор ресурса для строки, отображаемой на каждой кнопке. Второй Int - фактическое количество кексов.

Мы передадим второе свойство выбранной пары при вызове функции `onNextButtonClicked()`.

- Найдите пустое лямбда-выражение для параметра `onClick` кнопки `SelectQuantityButton`.

```
quantityOptions.forEach { item ->
    SelectQuantityButton(
        labelResourceId = item.first,
        onClick = {}
    )
}
```

- Внутри лямбда-выражения вызовите `onNextButtonClicked`, передав `item.second` - количество кексов.

```
quantityOptions.forEach { item ->
    SelectQuantityButton(
        labelResourceId = item.first,
        onClick = { onNextButtonClicked(item.second) }
    )
}
```

- Добавьте обработчики кнопок в `SelectOptionScreen` Ниже параметра `onSelectionChanged` композитного экрана `SelectOptionScreen` в `SelectOptionScreen.kt` добавьте параметр `onCancelButtonClicked` типа `() -> Unit` со значением по умолчанию `{}`.

```
@Composable
fun SelectOptionScreen(
    subtotal: String,
    options: List<String>,
    onSelectionChanged: (String) -> Unit = {},
    onCancelButtonClicked: () -> Unit = {},
    modifier: Modifier = Modifier
)
```

- Ниже параметра `onCancelButtonClicked` добавьте еще один параметр типа `() -> Unit` под названием `onNextButtonClicked` со значением по умолчанию `{}`.

```
@Composable
fun SelectOptionScreen(
    subtotal: String,
    options: List<String>,
    onSelectionChanged: (String) -> Unit = {},
    onCancelButtonClicked: () -> Unit = {},
    onNextButtonClicked: () -> Unit = {},
    modifier: Modifier = Modifier
)
```

- Передайте параметр `onCancelButtonClicked` для параметра `onClick` кнопки отмены.

```
OutlinedButton(
    modifier = Modifier.weight(1f),
    onClick = onCancelButtonClicked
) {
    Text(stringResource(R.string.cancel))
}
```

- Передайте параметр `onNextButtonClicked` для параметра `onClick` следующей кнопки.

```
Button(
    modifier = Modifier.weight(1f),
    enabled = selectedValue.isNotEmpty(),
    onClick = onNextButtonClicked
) {
    Text(stringResource(R.string.next))
}
```

- Добавьте обработчики кнопок на экран сводки Наконец, добавьте функции обработчика кнопок для кнопок `Cancel` и `Send` на экране сводки.

- В составном экране `OrderSummaryScreen` в файле `SummaryScreen.kt` добавьте параметр `onCancelButtonClicked` типа `() -> Unit`.

```
@Composable
fun OrderSummaryScreen(
    orderUiState: OrderUiState,
    onCancelButtonClicked: () -> Unit,
    modifier: Modifier = Modifier
){
    ...
}
```

- Добавьте еще один параметр типа `(String, String) -> Unit` и назовите его `onSendButtonClicked`.

```
@Composable
fun OrderSummaryScreen(
    orderUiState: OrderUiState,
    onCancelButtonClicked: () -> Unit,
    onSendButtonClicked: (String, String) -> Unit,
    modifier: Modifier = Modifier
){
    ...
}
```

- Композит `OrderSummaryScreen` теперь ожидает значений для `onSendButtonClicked` и `onCancelButtonClicked`. Найдите `OrderSummaryPreview`, передайте пустое тело лямбды с двумя параметрами String в `onSendButtonClicked` и пустое тело лямбды в параметры `onCancelButtonClicked`.

```
@Preview
@Composable
```

```
fun OrderSummaryPreview() {
    CupcakeTheme {
        OrderSummaryScreen(
            orderUiState = OrderUiState(0, "Test", "Test", "$300.00"),
            onSendButtonClicked = { subject: String, summary: String -> },
            onCancelButtonClicked = {},
            modifier = Modifier.fillMaxHeight()
        )
    }
}
```

- Передайте `onSendButtonClicked` для параметра `onClick` кнопки «Отправить». Передайте `newOrder` и `orderSummary`, две переменные, определенные ранее в `OrderSummaryScreen`. Эти строки содержат фактические данные, которыми пользователь может поделиться с другим приложением.

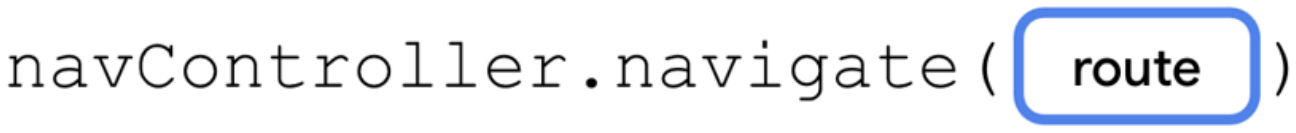
```
Button(
    modifier = Modifier.fillMaxWidth(),
    onClick = { onSendButtonClicked(newOrder, orderSummary) }
) {
    Text(stringResource(R.string.send))
}
```

- Передайте `onCancelButtonClicked` для параметра `onClick` кнопки «Отмена».

```
OutlinedButton(
    modifier = Modifier.fillMaxWidth(),
    onClick = onCancelButtonClicked
) {
    Text(stringResource(R.string.cancel))
}
```

Переход к другому маршруту

- Чтобы перейти к другому маршруту, просто вызовите метод `navigate()` на экземпляре `NavHostController`.



Метод `navigate` принимает единственный параметр: строку, соответствующую маршруту, определенному в вашем `NavHost`. Если маршрут совпадает с одним из вызовов функции `composable()` в `NavHost`, приложение переходит на этот экран.

Вы будете передавать функции, вызывающие `navigate()`, когда пользователь нажимает кнопки на экранах Start, Flavor и Pickup.

- В файле `CupcakeScreen.kt` найдите вызов функции `composable()` для начального экрана. Для параметра `onNextButtonClicked` передайте лямбда-выражение.

```
StartOrderScreen(
    quantityOptions = DataSource.quantityOptions,
    onNextButtonClicked = {
    }
)
```

Помните свойство Int, переданное в эту функцию для количества кексов? Перед переходом к следующему экрану необходимо обновить модель представления, чтобы приложение отображало правильный промежуточный итог.

- Вызовите `setQuantity` на модели представления, передав в нее.

```
onNextButtonClicked = {
    viewModel.setQuantity(it)
}
```

- Вызовите `navigate()` на `navController`, передав `CupcakeScreen.Flavor.name` для маршрута.

```
onNextButtonClicked = {
    viewModel.setQuantity(it)
    navController.navigate(CupcakeScreen.Flavor.name)
}
```

Для параметра `onNextButtonClicked` на экране вкуса просто передайте лямбду, которая вызывает `navigate()`, передавая `CupcakeScreen.Pickup.name` для маршрута.

```
composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(
        subtotal = uiState.price,
        onNextButtonClicked = { navController.navigate(CupcakeScreen.Pickup.name)
},
        options = DataSource.flavors.map { id -> context.resources.getString(id)
},
        onSelectionChanged = { viewModel.setFlavor(it) },
        modifier = Modifier.fillMaxHeight()
    )
}
```

- Передайте пустую лямбду для `onCancelButtonClicked`, которую вы реализуете далее.

```
SelectOptionScreen(
    subtotal = uiState.price,
    onNextButtonClicked = { navController.navigate(CupcakeScreen.Pickup.name) },
    onCancelButtonClicked = {},
    options = DataSource.flavors.map { id -> context.resources.getString(id) },
    onSelectionChanged = { viewModel.setFlavor(it) },
    modifier = Modifier.fillMaxHeight()
)
```

- Для параметра `onNextButtonClicked` на экране пикапа передайте лямбду, которая вызывает `navigate()`, передавая `CupcakeScreen.Summary.name` для маршрута.

```
composable(route = CupcakeScreen.Pickup.name) {
    SelectOptionScreen(
        subtotal = uiState.price,
        onNextButtonClicked = { navController.navigate(CupcakeScreen.Summary.name)
},
        options = uiState.pickupOptions,
        onSelectionChanged = { viewModel.setDate(it) },
        modifier = Modifier.fillMaxHeight()
    )
}
```

- Снова передайте пустую лямбду для `onCancelButtonClicked()`.

```
SelectOptionScreen(
    subtotal = uiState.price,
    onNextButtonClicked = { navController.navigate(CupcakeScreen.Summary.name) },
    onCancelButtonClicked = {},
    options = uiState.pickupOptions,
    onSelectionChanged = { viewModel.setDate(it) },
    modifier = Modifier.fillMaxHeight()
)
```

- Для экрана `OrderSummaryScreen` передайте пустые лямбды для `onCancelButtonClicked` и `onSendButtonClicked`. Добавьте параметры для темы и резюме, передаваемые в `onSendButtonClicked`, которые вы скоро реализуете.

```
composable(route = CupcakeScreen.Summary.name) {
    OrderSummaryScreen(
        orderUiState = uiState,
        onCancelButtonClicked = {},
        onSendButtonClicked = { subject: String, summary: String ->
```

```
        },
        modifier = Modifier.fillMaxHeight()
    )
}
```

Теперь вы должны иметь возможность перемещаться по каждому экрану вашего приложения. Обратите внимание, что при вызове `navigate()` экран не только меняется, но и помещается поверх стека back. Кроме того, при нажатии системной кнопки «Назад» вы можете вернуться к предыдущему экрану.

Приложение помещает каждый экран в стек поверх предыдущего, а кнопка «Назад» может их удалить. История экранов от начального пункта назначения внизу до самого верхнего экрана, который только что был показан, называется стопкой назад.

Возврат к начальному экрану В отличие от системной кнопки «Назад», кнопка «Отмена» не возвращает на предыдущий экран. Вместо этого она должна удалить все экраны из заднего стека и вернуться на начальный экран.

Это можно сделать, вызвав метод `popBackStack()`.

Метод `popBackStack()` имеет два обязательных параметра.

- **маршрут**: Строка, представляющая маршрут пункта назначения, к которому вы хотите вернуться. включительно: Булево значение, которое, если равно true, также удаляет указанный маршрут. Если значение равно false, `popBackStack()` удалит все пункты назначения поверх начального пункта назначения, но не включая его, оставив его самым верхним экраном, видимым пользователю. Когда пользователь нажимает кнопку «Отмена» на любом из экранов, приложение сбрасывает состояние в модели представления и вызывает `popBackStack()`. Сначала вы реализуете метод для этого, а затем передадите ему соответствующий параметр на всех трех экранах с кнопками Cancel.
- После функции `CupcakeApp()` определите частную функцию `cancelOrderAndNavigateToStart()`.

```
private fun cancelOrderAndNavigateToStart() {
}
```

- Добавьте два параметра: viewModel типа OrderViewModel и navController типа NavHostController.

```
private fun cancelOrderAndNavigateToStart(
    viewModel: OrderViewModel,
    navController: NavHostController
```

```
) {
}
```

- В теле функции вызовите `resetOrder()` для viewModel.

```
private fun cancelOrderAndNavigateToStart(
    viewModel: OrderViewModel,
    navController: NavHostController
) {
    viewModel.resetOrder()
}
```

- Вызовите `popBackStack()` на navController, передав `CupcakeScreen.Start.name` для маршрута и false для включения.

```
private fun cancelOrderAndNavigateToStart(
    viewModel: OrderViewModel,
    navController: NavHostController
) {
    viewModel.resetOrder()
    navController.popBackStack(CupcakeScreen.Start.name, inclusive = false)
}
```

- В составной части `CupcakeApp()` передайте `cancelOrderAndNavigateToStart` для параметров `onCancelButtonClicked` двух составных частей SelectOptionScreen и составной части OrderSummaryScreen.

```
composable(route = CupcakeScreen.Start.name) {
    StartOrderScreen(
        quantityOptions = DataSource.quantityOptions,
        onNextButtonClicked = {
            viewModel.setQuantity(it)
            navController.navigate(CupcakeScreen.Flavor.name)
        },
        modifier = Modifier
            .fillMaxSize()
            .padding(dimensionResource(R.dimen.padding_medium))
    )
}
composable(route = CupcakeScreen.Flavor.name) {
    val context = LocalContext.current
    SelectOptionScreen(
        subtotal = uiState.price,
        onNextButtonClicked = { navController.navigate(CupcakeScreen.Pickup.name)
},
        onCancelButtonClicked = {
            cancelOrderAndNavigateToStart(viewModel, navController)
```

```
            },
            options = DataSource.flavors.map { id -> context.resources.getString(id)
    },
            onSelectionChanged = { viewModel.setFlavor(it) },
            modifier = Modifier.fillMaxHeight()
        )
    }
    composable(route = CupcakeScreen.Pickup.name) {
        SelectOptionScreen(
            subtotal = uiState.price,
            onNextButtonClicked = { navController.navigate(CupcakeScreen.Summary.name)
    },
            onCancelButtonClicked = {
                cancelOrderAndNavigateToStart(viewModel, navController)
            },
            options = uiState.pickupOptions,
            onSelectionChanged = { viewModel.setDate(it) },
            modifier = Modifier.fillMaxHeight()
        )
    }
    composable(route = CupcakeScreen.Summary.name) {
        OrderSummaryScreen(
            orderUiState = uiState,
            onCancelButtonClicked = {
                cancelOrderAndNavigateToStart(viewModel, navController)
            },
            onSendButtonClicked = { subject: String, summary: String ->

            },
            modifier = Modifier.fillMaxHeight()
       )
    }
```

- Запустите приложение и проверьте, что нажатие кнопки «Отмена» на любом из экранов возвращает пользователя к первому экрану.

Переход в другое приложение

Итак, вы узнали, как перейти на другой экран в вашем приложении и как вернуться на главный экран. Осталось сделать еще один шаг, чтобы реализовать навигацию в приложении `Cupcake`. На экране сводки заказов пользователь может отправить свой заказ в другое приложение. Этот выбор вызывает `ShareSheet` - компонент пользовательского интерфейса, который занимает нижнюю часть экрана и показывает варианты совместного использования.

Этот элемент пользовательского интерфейса не является частью приложения `Cupcake`. На самом деле он предоставляется операционной системой Android. Системный пользовательский интерфейс, такой как экран совместного доступа, не вызывается вашим `navController`. Вместо этого вы используете нечто, называемое намерением (intent)

Намерение - это запрос к системе на выполнение какого-либо действия, обычно это представление новой активности. Существует множество различных намерений, и мы рекомендуем вам обратиться к

документации за полным списком. Однако нас интересует намерение под названием `ACTION_SEND`. Вы можете предоставить этому намерению некоторые данные, например строку, и представить соответствующие действия по обмену этими данными.

Основной процесс настройки намерения выглядит следующим образом:

- Создайте объект намерения и укажите намерение, например ACTION_SEND.
- Укажите тип дополнительных данных, отправляемых вместе с намерением. Для простого текста можно использовать «text/plain», хотя возможны и другие типы, например «image/» *или «video/»*.
- Передайте намерению любые дополнительные данные, например текст или изображение, которыми нужно поделиться, вызвав метод putExtra(). Это намерение примет два дополнительных параметра: EXTRA_SUBJECT и EXTRA_TEXT.
- Вызовите метод startActivity() контекста, передав в него активность, созданную на основе намерения.

Мы расскажем вам о том, как создать намерение `share action`, но процесс не отличается от других типов намерений. В будущих проектах рекомендуем вам обращаться к документации по мере необходимости, чтобы узнать о конкретном типе данных и необходимых дополнениях.

Выполните следующие шаги, чтобы создать намерение отправить заказ кекса в другое приложение:

- В файле `CupcakeScreen.kt`, расположенном ниже композита `CupcakeApp`, создайте приватную функцию `shareOrder()`.

```
private fun shareOrder()
```

- Добавьте параметр с именем context типа Context.

```
import android.content.Context

private fun shareOrder(context: Context) {
}
```

- Добавьте два параметра String: subject и summary. Эти строки будут отображаться на листе действий по обмену.

```
private fun shareOrder(context: Context, subject: String, summary: String) {
}
```

- В теле функции создайте Intent с именем intent и передайте `Intent.ACTION_SEND` в качестве аргумента.

```
import android.content.Intent
```

```
val intent = Intent(Intent.ACTION_SEND)
```

Поскольку вам нужно настроить этот объект `Intent` только один раз, вы можете сделать следующие несколько строк кода более краткими с помощью функции `apply()`, о которой вы узнали в одном из предыдущих уроков.

- Вызовите `apply()` для только что созданного интента и передайте лямбда-выражение.

```
val intent = Intent(Intent.ACTION_SEND).apply {

}
```

- В теле лямбды установите тип «text/plain». Поскольку вы делаете это в функции, передаваемой в `apply()`, вам не нужно ссылаться на идентификатор объекта, intent.

```
val intent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
}
```

- Вызовите функцию putExtra(), передав тему для EXTRA_SUBJECT.

```
val intent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
    putExtra(Intent.EXTRA_SUBJECT, subject)
}
```

- Вызовите метод `startActivity()` контекста.

```
context.startActivity(

)
```

- В лямбде, переданной в `startActivity()`, создайте активность из намерения, вызвав метод класса `createChooser()`. Передайте намерение в качестве первого аргумента и строковый ресурс new_cupcake_order.

```
context.startActivity(
    Intent.createChooser(
        intent,
        context.getString(R.string.new_cupcake_order)
    )
)
```

- В композите `CupcakeApp` в вызове `composable()` для `CucpakeScreen.Summary.name` получите ссылку на объект контекста, чтобы передать его в функцию `shareOrder()`.

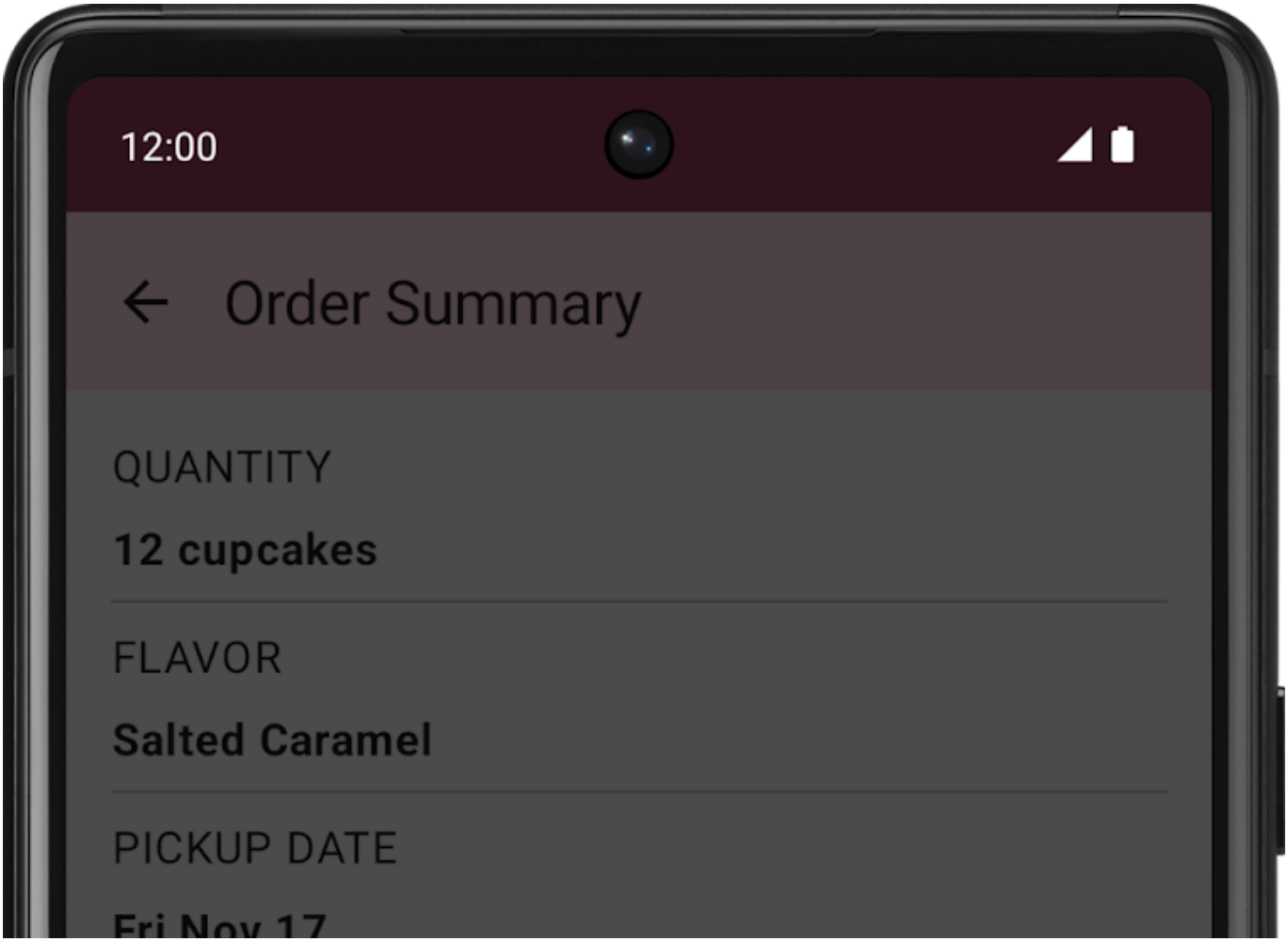
```
composable(route = CupcakeScreen.Summary.name) {
    val context = LocalContext.current

    ...
}
```

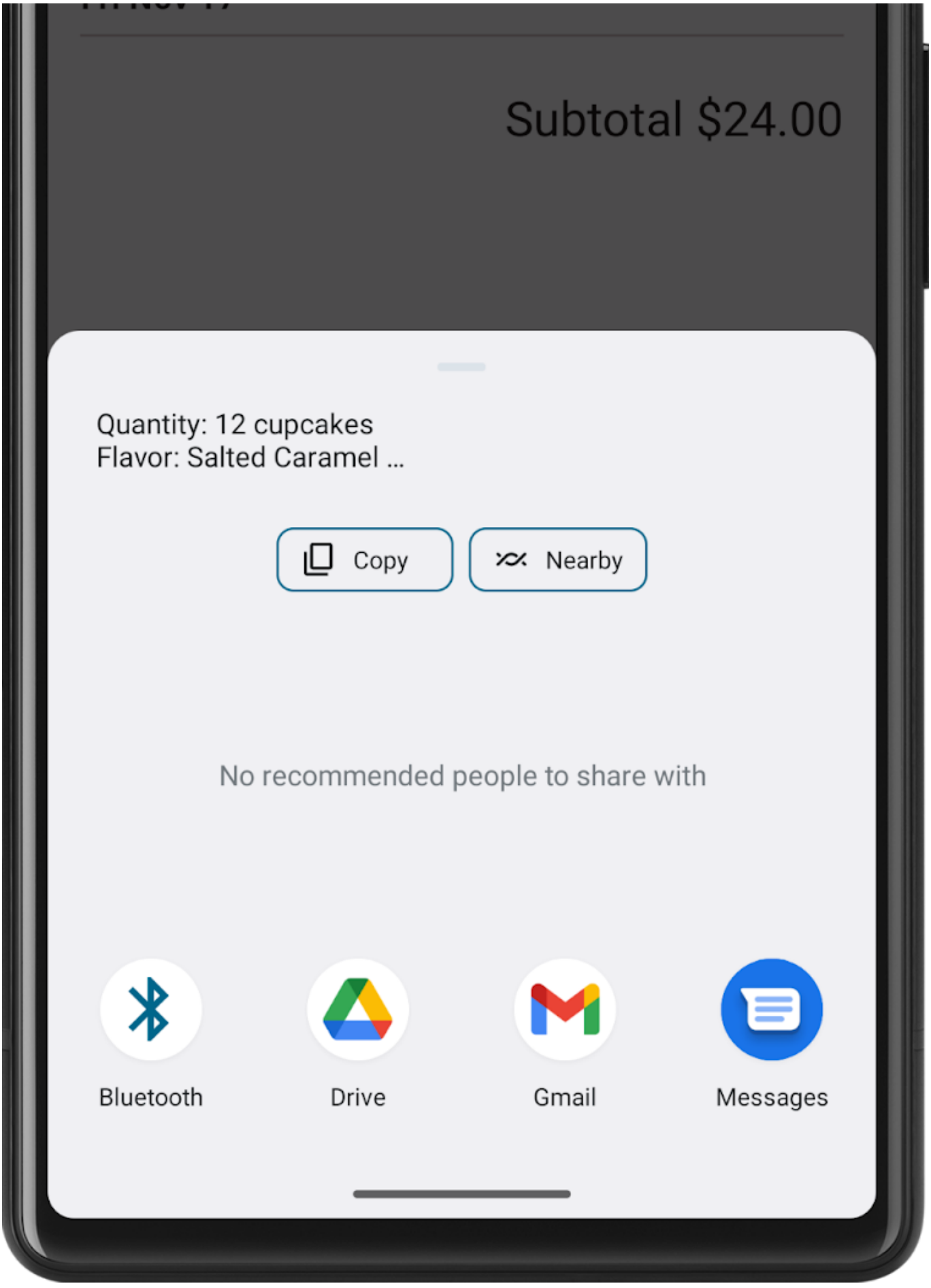
- В теле лямбды `onSendButtonClicked()` вызовите `shareOrder()`, передав в качестве аргументов контекст, тему и резюме.

```
onSendButtonClicked = { subject: String, summary: String ->
    shareOrder(context, subject = subject, summary = summary)
}
```

- Запустите приложение и перемещайтесь по экранам.

Когда вы нажмете «Отправить заказ в другое приложение», на нижнем листе должны появиться такие действия совместного доступа, как «Передача сообщений» и «Bluetooth», а также тема и резюме, которые вы указали в качестве дополнительных сведений.

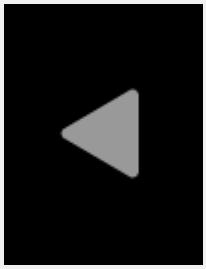
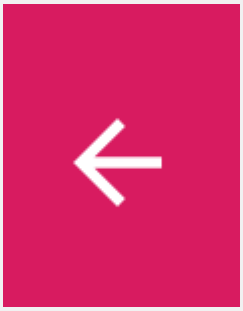




{style="width:400px"}

Заставьте панель приложения реагировать на навигацию

Несмотря на то, что ваше приложение функционирует и позволяет переходить к любому экрану и обратно, на скриншотах в начале этого урока все еще чего-то не хватает. Панель приложений не реагирует на навигацию автоматически. Заголовок не обновляется, когда приложение переходит к новому маршруту, и не отображает кнопку «Вверх» перед заголовком, когда это необходимо.

> Примечание: Системная кнопка «Назад» предусмотрена операционной системой Android и расположена в нижней части экрана.  Кнопка «Вверх», с другой стороны, находится в AppBar вашего приложения.  В контексте вашего приложения и кнопка «Назад», и кнопка «Вверх» выполняют одну и ту же функцию - навигацию к предыдущему экрану.

Стартовый код включает в себя компонент для управления `AppBar` под названием `CupcakeAppBar`. Теперь, когда вы реализовали навигацию в приложении, вы можете использовать информацию из StackBack, чтобы отобразить правильный заголовок и показать кнопку Up, если это необходимо. Составной `CupcakeAppBar` должен знать о текущем экране, чтобы заголовок обновлялся соответствующим образом.

- В перечисление `CupcakeScreen` в `CupcakeScreen.kt` добавьте параметр типа Int с именем title, используя аннотацию `@StringRes`.

```
import androidx.annotation.StringRes

enum class CupcakeScreen(@StringRes val title: Int) {
    Start,
    Flavor,
    Pickup,
    Summary
}
```

- Добавьте значение ресурса для каждого случая перечисления, соответствующее тексту заголовка для каждого экрана. Используйте app_name для начального экрана, choose_flavor для экрана Flavor, choose_pickup_date для экрана Pickup и order_summary для экрана Summary.

```
enum class CupcakeScreen(@StringRes val title: Int) {
    Start(title = R.string.app_name),
```

```
    Flavor(title = R.string.choose_flavor),
    Pickup(title = R.string.choose_pickup_date),
    Summary(title = R.string.order_summary)
}
```

- Добавьте параметр `currentScreen` типа `CupcakeScreen` в составной `CupcakeAppBar`.

```
fun CupcakeAppBar(
    currentScreen: CupcakeScreen,
    canNavigateBack: Boolean,
    navigateUp: () -> Unit = {},
    modifier: Modifier = Modifier
)
```

- Внутри `CupcakeAppBar` замените жестко закодированное название приложения на заголовок текущего экрана, передав параметр `currentScreen.title` в вызов `stringResource()` для параметра title в `TopAppBar`.

```
TopAppBar(
    title = { Text(stringResource(currentScreen.title)) },
    modifier = modifier,
    navigationIcon = {
        if (canNavigateBack) {
            IconButton(onClick = navigateUp) {
                Icon(
                    imageVector = Icons.Filled.ArrowBack,
                    contentDescription = stringResource(R.string.back_button)
                )
            }
        }
    }
)
```

Кнопка «Вверх» должна отображаться только в том случае, если на заднем стеке есть композитный экран. Если в приложении нет экранов на заднем стеке - показан экран `StartOrderScreen`, - то кнопка «Вверх» не должна отображаться. Чтобы проверить это, вам нужна ссылка на задний стек.

- В композите CupcakeApp под переменной navController создайте переменную backStackEntry и вызовите метод currentBackStackEntryAsState()в navController с помощью делегата by.

```
import androidx.navigation.compose.currentBackStackEntryAsState

@Composable
fun CupcakeApp(
    viewModel: OrderViewModel = viewModel(),
    navController: NavHostController = rememberNavController()
){
```

```
    val backStackEntry by navController.currentBackStackEntryAsState()

    ...
}
```

- Преобразуйте заголовок текущего экрана в значение `CupcakeScreen`. Под переменной backStackEntry создайте переменную val с именем currentScreen, равную результату вызова функции valueOf() класса CupcakeScreen, и передайте в нее маршрут назначения backStackEntry. Используйте оператор elvis, чтобы задать значение по умолчанию для CupcakeScreen.Start.name.

```
val currentScreen = CupcakeScreen.valueOf(
    backStackEntry?.destination?.route ?: CupcakeScreen.Start.name
)
```

- Передайте значение переменной `currentScreen` в одноименный параметр композита `CupcakeAppBar`.

```
CupcakeAppBar(
    currentScreen = currentScreen,
    canNavigateBack = false,
    navigateUp = {}
)
```

Пока за текущим экраном в заднем стеке есть экран, кнопка «Вверх» должна отображаться. Вы можете использовать булево выражение, чтобы определить, должна ли отображаться кнопка Up:

- Для параметра canNavigateBack передайте булево выражение, проверяющее, не равно ли свойство previousBackStackEntry контроллера navController значению null.

```
canNavigateBack = navController.previousBackStackEntry != null,
```

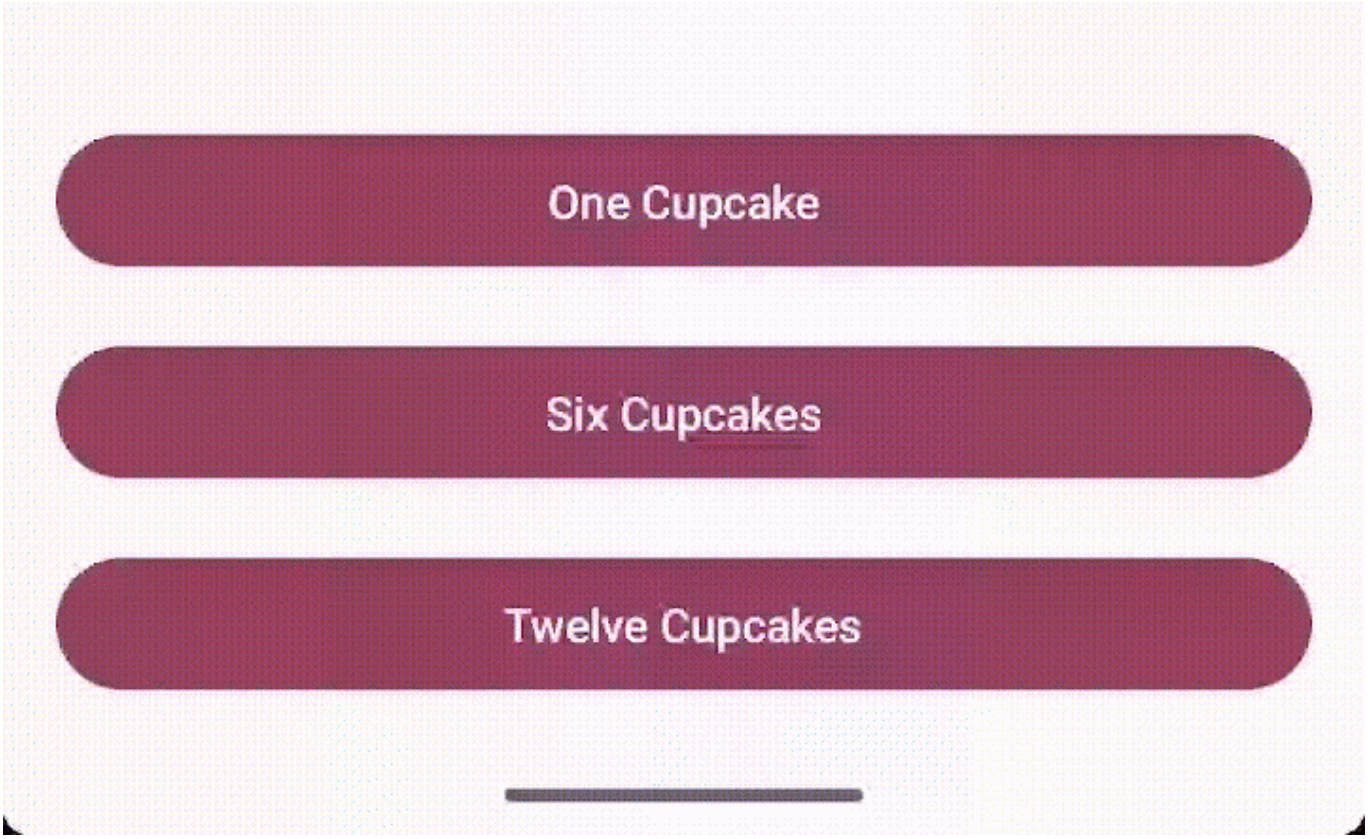
- Чтобы перейти к предыдущему экрану, вызовите метод navigateUp() контроллера navController.

```
navigateUp = { navController.navigateUp() }
```

- Запустите ваше приложение. Обратите внимание, что заголовок `AppBar` теперь обновляется и отражает текущий экран. При переходе на экран, отличный от `StartOrderScreen`, должна появиться кнопка Up, которая вернет вас на предыдущий экран.

Cupcake

Order Cupcakes



Получение кода решения

Чтобы загрузить код готового урока, вы можете воспользоваться этими командами git:

```
$ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-cupcake.git
$ cd basic-android-kotlin-compose-training-cupcake
$ git checkout navigation
```

Резюме

Вы только что перешли от простого одноэкранного приложения к сложному многоэкранному приложению, использующему компонент `Jetpack Navigation` для перемещения по нескольким экранам. Вы определили маршруты, обработали их в `NavHost` и использовали параметры типа функции, чтобы отделить логику навигации от отдельных экранов. Вы также узнали, как отправлять данные в другое приложение с помощью `intents` , а также настраивать панель приложения в ответ на навигацию. В следующих разделах вы продолжите использовать эти навыки, работая над несколькими другими многоэкранными приложениями, сложность которых возрастает.