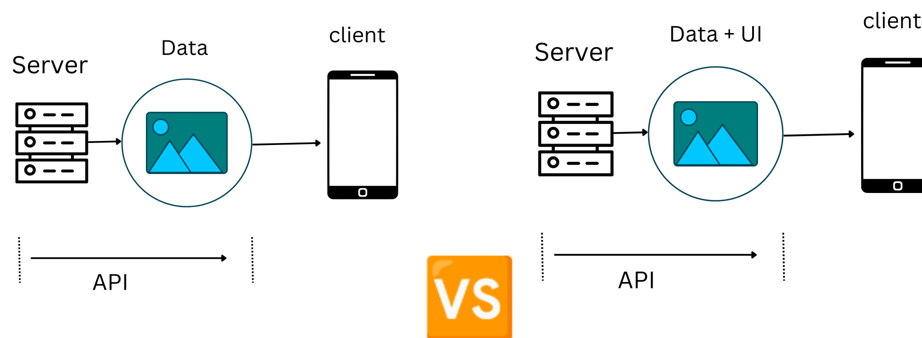
Server-Driven UI for Mobile: Getting Started

Server-Driven UI is a software Design pattern that I have yet to see many people explicitly talk about throughout most of my career as a software engineer. However, there have been very few talks about this trend at some conferences, but so far, I have yet to see any of them go hands-on with its implementation.

In this article, I will discuss all that is needed to be known about Server-Driven UI, we will use a simple screen that we will be building together as a case study.

What is Server Driven UI

To put it simply, Server Driven UI is a software design pattern that lets the server drive the rendering of UI components. To put this in a better context I will highlight how this approach is different from the traditional client-driven design.



The picture above depicts the difference between the traditional way of building mobile apps(Client Driven UI) and Sever-Driven UI.

With Client Driven UI the server is only responsible for sending and receiving data. and the Client is responsible for handling UI logic. In this pattern, the server has no control over what and how your components are being rendered. But, what if we let the UI tell the Client what to render and how it should be rendered?
And you might be curious, why should we even do that? Well, hold my coffee, to do enough justice to this question we need to point out some of the shortcomings of client-driven UI

Shortcomings of Client-Driven UI

While Server Driven UI might not be a silver bullet to solve all of the shortcomings of the client-driven UI, it does solve some of the problems I will be mentioning In the article up to a very great extent.

1.) Inconsistency & duplication of effort across platforms: when the same version of an app is live on multiple platforms, there might be some inconsistency that can arise when you make changes to UI components, to maintain consistency across Clients (Platforms) you will have to alter code in each Client, which is like a duplication of effort that can take a

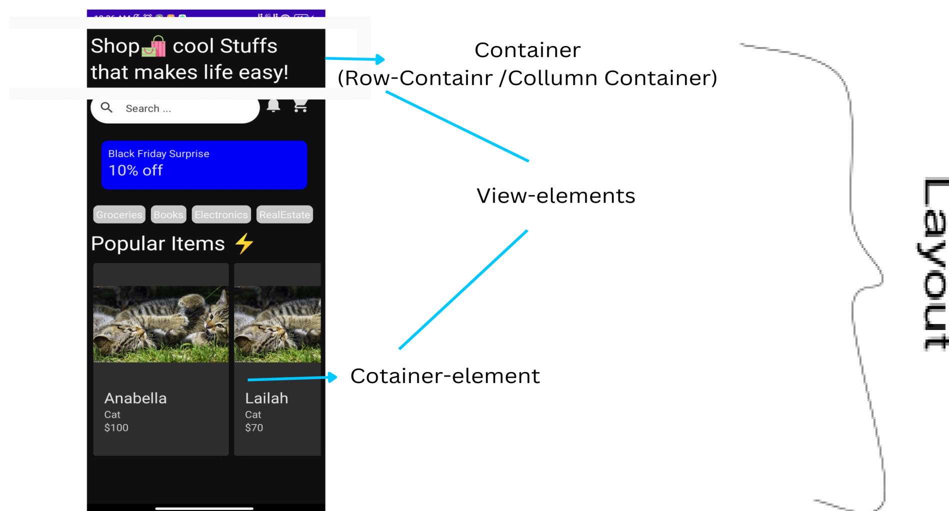
toll on productivity. There is also no guarantee that those changes will look and feel consistent across clients.

2.) Versioning: When you push a new version of your app with UI changes, there is usually no clear strategy to make sure every user does update to the latest version of your app, with server-driven UI you don't have to bother about that, you already will be confident that every user gets the latest changes.

3.) A/B Testing: With Client Driven UI it's usually hard to perform A/B mostly due to the problems I highlighted above, But Server-Driven UI makes it easier, as you can guarantee that certain users get a version of your app at a particular point in time, which foster A/B Testing practices

Building the Server side API:

Before building a server-driven UI API, It's important to have a Design Language/ Design System in place, that way the server and the clients are in sync with possible components that can be rendered.



The Design system:
For the app shown above which is what we will be building together in this guide, we will have a straightforward and simple design system that I call the L -> C -> C System. LCC basically stands for Layout, Container, and Container-Elements.

Layouts: Layout could be anything from an entire screen to a section of the screen. A layout contains one or multiple Containers.
Containers: There are two types of containers in our design system, which are categorized based on how elements are arranged in them. We have the Row-Container which arranges elements horizontally and the Column-Container which arranges contents vertically.

Container-Elements: There are different kinds of elements in our design system, we have the typography, Image, Card, Button, and the Input-Text.

While View-Element is a collective name for every item that appears on the screen

For the server-side API, we will be using GraphQl as our API design protocol against the fundamental Rest Protocol
So, why GraphQL? Specific bottlenecks can be encountered while building Server Driven UI API with Rest protocol, which is what GraphQL is designed to optimize, some of these problems are mentioned in [this](#) article, however, one reason why I think GraphQL is a better candidate for server-driven UI API is the fact that it returns concrete Object, so there is no need for JSON serializing and deserializing.

We will also be using Typescript in this guide as that will allow us to leverage the types feature, that comes with Typescript

Since we have established the basics of Server-driven UI and how it might be beneficial, now let's get our hands dirty with some code !! 🚀

NOTE: This isn't a GraphQL crash course so I won't cover the basics of graphQL, so I will recommend that you understand the basics of GraphQL before getting started with this tutorial. Some basic knowledge of typescript and nodeJs will also go a long way, however, you don't necessarily have to know NodeJs before getting started

For this tutorial, I already prepared a barebone project that you can get started with, you only just have to clone the [project](#) and install all the dependencies by using `npm ci`

The API - Our Design System:
Our design system will be defined in graphQL schemas from which a typescript type will be generated. Let's first define a schema for our ***Container***, to do that navigate to `***The Server/graphql/container.graphql***` from the root directory and input this code:

```
"""
Container is used for holding many different elements.
It can hold typography items like headings, and body text, but
also can hold UI View Elements like Cards, Banners etc.
"""
type Container {
   containerType: ContainerType!
   elements: [ContainerElement!]
}
enum ContainerType {
```

```
   ROW
   COLUMN
}
```

Now, from the root directory, navigate to the `graphql` directory and create a new file called `***view-element.graphql*** ` inside of that file, put the code:

```
"""
All composable view-elements are to be placed here
"""
union ViewElement = Container
```

With that in place, now run `**npm run generate-types**` in your terminal, once that is done then check the `***The Server/src/types.ts***` file from the root directory, you should see actual typescript types that represent our UI elements which were generated from our graphql Schema.

With that in place we need to create a graphQL schema that represents our query, to do that navigate to the `The Server/graphql/queries.graphql` directory from the root directory and put this code

```
type Query{
 sampleScreen : SampleScreenView
}
```

The next piece is that we need to find a way to tell API what and how Object (in our case, UI elements)to return and when to return it.

And that's when the Resolvers come in, to create our first resolver, let's navigate to `***The Server/*****src/resolvers/view-element-resolver**` and put this code:

```
export const ViewElementResolver = {
 ViewElement:{
   __resolveType(obj : any){
     if (obj.elements && obj.containerType){
       return 'Container'
     }
     return null;
   }
 }
}
```

With that in place now let’s go ahead and query for our first screen!:

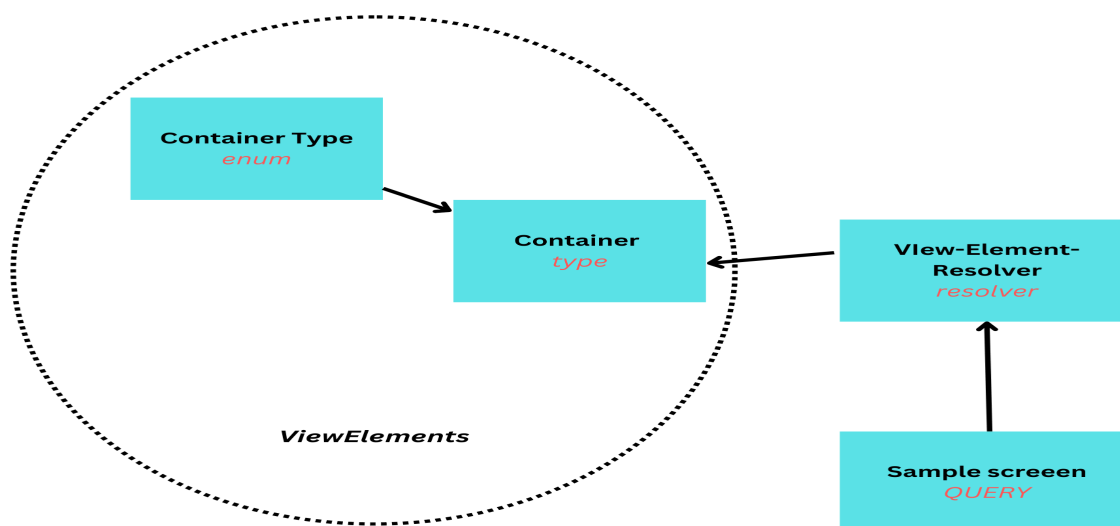
To do that navigate to `**The Server/src/queries/sample-screen-query**` from the root directory and input this code:

```
export  const sampleScreenQuery = {
 sampleScreen : ()=> {
  return {
    elements: [
      {
        containerType : 'COLUMN',
        elements:[

        ]
      }
    ]
  }
 }
}
```

Now, run `npm start`, you should have something like this in your terminal:

```
√ Parse Configuration
√ Generate outputs
Server is ready at http://localhost:4000/graphql
```

With that our server-driven UI server is up and running with a screen and an empty container! We will be adding other components to this soon, but let me explain with a brief overview of what we have done so far,



The picture above describes an overview of what we have just achieved with our code, the sample screen query is where we put all our Containers in the order we want it, while view-element Resolver tells our API which objects to be returned based on the parameter given by the sample-screen query(*At the moment we only have one container*)

This is pretty much the pattern our API will be following, now let's go ahead and add other views-elements like Banners, Cards, etc.

To add our view elements we will follow the same pattern of adding graphQL schemas to our graphQL directory. Now, navigate to the `The Server/graphql` from the root directory create a graphq file and call it `other-view-elements.graphql` and put this code:

```
"""
BOX
"""
type Box {
 _debugColor: Color
 width: Int
 height: Int
}

"""
BUTTON
"""
enum ButtonVariant {
 TEXT
 CONTAINED
 OUTLINED
}

enum ButtonTheme {
 PRIMARY
 SECONDARY
 SUCCESS
 ERROR
}

enum ButtonSize {
 SMALL
 MEDIUM
 LARGE
}

interface Buttons {
 icon: String
 label: String
```

```
 action: Action
 disabled: Boolean!
 buttonSize: ButtonSize!
}

type Button implements Buttons {
 icon: String
 label: String
 action: Action
 disabled: Boolean!
 disableElevation: Boolean!
 buttonVariant: ButtonVariant!
 buttonTheme: ButtonTheme!
 buttonSize: ButtonSize!
}

type IconButton implements Buttons{
 label: String
 icon: String
 action: Action
 disabled: Boolean!
 buttonSize: ButtonSize!
}

type FavouriteButton implements Buttons {
 label: String
 signal: Signal
 icon: String
 action: Action
 disabled: Boolean!
 buttonSize: ButtonSize!
}
"""
CARD
"""
type Card {
 signal: Signal
 primary: String!
 secondaries: [String!]
 media: Image
 content: [String!]
 links: [Buttons!]
 action: Action
}
"""
CHIP
"""
```

```
type Chip{
 chipLabel : String,
}
"""
COLUMN
"""
type ColumnLayout {
 columns: [Column]
}

enum ColumnAlignment {
 LEFT
 RIGHT
 CENTER
}

interface Column {
 align: ColumnAlignment!
}
"""
IMAGE
"""
type Image {
 url: String!
 alt: String!
 width: Int
 height: Int
 valueType: ImageValueType
}

enum ImageValueType {
 PIXEL
 PERCENTAGE
}

"""
SIGNAL
"""
interface EmitSignals {
 emitSignals: [EmitSignal!]
}

type EmitSignal {
 signal: Signal!
 values: [SignalValuePair!]!
}
```

```
enum SignalValuePairKey {
 ICON
 CONTENT
 PRIMARY
}

type SignalValuePair {
 key: SignalValuePairKey!,
 value: SignalValuePairValue!
}

type SignalStringValue {
 text: String!
}

type SignalArrayValue {
 prefix: [String!]
 suffix: [String!]
 array: [String!]!
}

union SignalValuePairValue = SignalStringValue | SignalArrayValue

type Signal {
 type: SignalType!
 reference: String
}

input SignalInput {
 type: SignalType!
 reference: String
}

enum SignalType {
 TOGGLE
 UPDATE
 TITLE
 ERROR
}

"""
TYPOGRAPHY
"""
type Typography {
 signal: Signal
 value: String!
 typographyVariant: TypographyVariant!
```

```
 typographyTheme: TypographyTheme!
}

enum TypographyTheme {
 PRIMARY
 SECONDARY
}

enum TypographyVariant {
 H1
 H2
 H3
 H4
 H5
 H6
 BODY1
 BODY2
 SUBTITLE1
 SUBTITLE2
 CAPTION
 OVERLINE
}
"""
ACTION
"""
union Action = EditNameSubmitAction | URLAction | FavouriteAction

type URLAction {
 url: String!
 description: String
}

type EditNameSubmitAction {
 inputIds: [String!]!
 emitSignal: EmitSignal!
}

type FavouriteAction {
 feedId: String!
 save: [EmitSignal!]
 unsave: [EmitSignal!]
}

"""
COLOR
"""
enum Color {
```

```
  PRIMARY
  SECONDARY
  ERROR
  WARNING
  INFO
  SUCCESS
}
```

Now run `npm run generate` from your terminal, you should have your full schema generated in the 'The server/src/graphql.schema' directory. With that in place, then you can run `npm run generate-types` this will generate actual types in the `types.ts` file as we did while building our container.

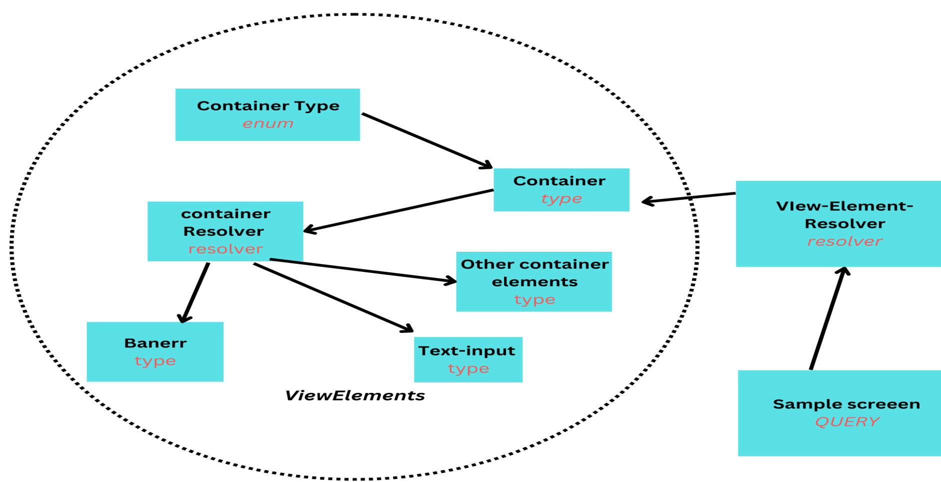
NB: Ideally, every View-element should have there own graphQL schema, this is just for the sake of this tutorial(*for reference you can checkout the final branch*).

Now let us create a resolver for our container-elements, to do that add this to code to `The Server/src/resolvers/container-resolver.ts`:

```
export const containerResolver = {
 ContainerElement :{
   __resolveType(obj: any){
     if (obj.typographyVariant && obj.value) {
       return 'Typography';
     }

     if (obj.primaryText && obj.secondaryText){
       return 'Banner'
     }

     if (obj.width || obj.height) {
       return 'Box';
     }

     if (obj.buttonVariant && obj.label) {
       return 'Button';
     }
     if (obj.url && obj.alt) {
       return 'Image';
     }
     if (obj.placeholder || obj.formId){
       return 'TextInput'
```

```
      }
      if (obj.primary) {
        return 'Card';
      }
      if (obj.chipLabel){
        return 'Chip'
      }
      if (obj.icon){
        return  'IconButton'
      }
      return  null
    }
  }
}
```

What this code does is it return View-element based on the parameter added to the query.



The diagram above shows a brief over of the moving parts in our API, The Container Resolver is responsible for deciding what element to be displayed based on the query passed in the sample screen.

Now, let's write our query! To write our screen query navigate to: `***The Server/src/query/sampleScreenQuery***` and insert this code:

```
export  const sampleScreenQuery = {
 sampleScreen : () =>{
   return{
     //  view elements
     elements: [
       {
         //container
```

```
    containerType : 'COLUMN',
    elements: [
      {
        typographyVariant: 'H3',
        value: 'Shop🛍️ cool Stuffs',
        typographyTheme: 'PRIMARY',
      },
      {
        typographyVariant: 'H3',
        value: 'that makes life easy!',
        typographyTheme: 'PRIMARY',
      },
    ],
  },
  {
    //  container
    containerType: 'ROW',
    elements: [
      //TextInput
      {
        formId: 'headingInput',
        placeholder: 'Search ...',
      },
      {
        icon: "notifications",
        action: null,
        disabled: false,
        disableElevation: false,
        buttonTheme: 'PRIMARY',
        buttonSize: 'MEDIUM',
      }
    ]
  },
  {
    //container
    containerType: 'COLUMN',
    elements:[
      {
        //Banner
        primaryText: '10% off',
        secondaryText : 'Black Friday Surprise'
      }
    ]
  },
  {
    //  container
    containerType: 'ROW',
```

```
        elements: [
          {
            chipLabel: "Groceries"
          },
          {
            chipLabel: "Books"
          },
          {
            chipLabel: "Electronics"
          },
          {
            chipLabel: "RealEstate"
          },
        ]
      },
      {
        containerType: 'COLUMN',
        elements: [
          {
            typographyVariant: 'H3',
            value: 'Popular Items ⚡',
            typographyTheme: 'PRIMARY',
          }
        ]
      },
      {
        //container
        containerType: 'ROW',
        elements: [
          // Card
          {
            primary: 'Media link card',
            secondaries: ['Secondary text'],
            links: [

            ],
            media: {
              url: 'https://picsum.photos/id/237/200/300',
              alt: 'Random image',
            },
          },

          {
            primary: 'Media link card',
            secondaries: ['Secondary text'],
            links: [
```

```
            ],
            media: {
              url: 'https://picsum.photos/500/400',
              alt: 'Random image',
            },
          },
        ],
      },
    ]
  }
 }
}
```

One more thing, before our API is up and running we need to let GraphQL know what elements Container elements can emit, to do this, now navigate to `***The Server/graphql***`` from the root directory and crate a file called view-element.graphQL then insert this code:

```
"""
All composable view-elements are to be placed here
"""
union ContainerElement = Card | Typography | Box | Button | IconButton | Image |
TextInput | Banner | Chip

union ViewElement = Container
```

This is code is responsible for telling our API what posible Object can be returned in from our Queries. If this concept isn't clear you can read more on graphQL unions here

Voilla!! 🎉 , Our API is ready !!

Now, let’s move to the next Phase!!
Building our Mobile APP,

Building the mobile app isn’t as trivial as building the Server API, especially if you have a design system in place. I already made a provision for our design system in the initial project so I will just walk through making the request and rendering the UI together.

First, we need to query our Components, for this, I will be using the fragment feature of GraphQL(not to be confused with android fragments), which allows us to make use of our components in multiple places. To do that navigate to
`TheAPP/app/src/main/graphql/fragments.graphql` and put this code

```
fragment container on Container {
   __typename
   containerType
   elements {
     ...card
     ...typography
     ...box
     ...button
     ...image
     ...textInput
     ...banner
     ...chip
     ...iconButton
   }
 }
fragment editNameContainer on EditNameContainer {
     __typename
     elements {
       ...textInput
       ...button
     }
}
fragment editNameSubmitAction on EditNameSubmitAction {
     __typename
     inputIds
     emitSignal {
         ...emitSignal
     }
 }

fragment urlAction on URLAction {
     __typename
     url
     description

}
fragment banner on Banner {
   primaryText
   secondaryText
}
 fragment box on Box {
    __typename
    _debugColor
    width
    height
 }
```

```
fragment button on Button {
   __typename
   label
   action {
     ...editNameSubmitAction
     ...urlAction
     ...favouriteAction
   }
   disabled
   disableElevation
   buttonVariant
   buttonTheme
   buttonSize
   icon
 }
fragment iconButton on IconButton{
    __typename
    label
    icon
    action{
     ...urlAction
    }
    disabled
    buttonSize
}

fragment card on Card {
    __typename
    primary
    secondaries
    content
    action {
      ...urlAction
    }
    links {
      ...button
      ...favouriteButton
    }
    media {
      ...image
    }
    signal {
      ...signal
    }
}

fragment chip on Chip{
```

```
  __typename
  chipLabel
 }

fragment emitSignal on EmitSignal {
     __typename
     signal {
       ...signal
     }
     values {
       ...signalValuePair
     }
   }


fragment favouriteAction on FavouriteAction {
     __typename
     feedId
     save {
       ...emitSignal
     }
     unsave {
       ...emitSignal
     }
  }

 fragment favouriteButton on FavouriteButton {
     __typename
     action {
       ...editNameSubmitAction
       ...urlAction
       ...favouriteAction
     }
     disabled
     buttonSize
     icon
     signal {
       ...signal
     }
  }

 fragment image on Image {
   __typename
   url
   alt
 }
```

```
 fragment sampleScreenView on SampleScreenView{
  elements{
   ...container
  }
 }

 fragment signal on Signal {
     __typename
     type
     reference
 }

   fragment signalValuePair on SignalValuePair {
     __typename
     key
     value {
       ...signalStringValue
       ...signalArrayValue
     }
   }

   fragment signalStringValue on SignalStringValue {
     __typename
     text
   }

fragment signalArrayValue on SignalArrayValue {
     __typename
     prefix
     suffix
     array
 }

 fragment textInput on TextInput {
     __typename
     formId
     placeholder
  }


 fragment typography on Typography {
     __typename
     typographyVariant
     typographyTheme
     value
  }
```

Ideally, each component should be queried in there own .graphql file (but for this sake of this article, let's just leave it this way).

NOTE: I will recommend readers take a crash course on graphQL if these concepts are not clear.

Now let's make our first query, to do that navigate to `TheAPP/app/src/main/graphql/com.example`, create a new file named `sample-screen.graphql`, and put this code:

```
query GetSampleScreenElementView {
sampleScreen{
 ...sampleScreenView
}
}
```

With that done build your app, if this is done successfully then you should have a code generated by Apollo-Kotlin, you can take a look at the generated code at `TheApp/app/build/generated/source/apollo` directory.

With that in place, now we are ready to build our UI.
Let's start first by creating our elementFactory Composable function. In case you might be wondering what our element factory method is, it's just a Kotlin function that tells our APP what element to return to display based on the objects returned by our remote API.
To write our elementFactory method navigate to the MainActivity of our app and put this function just below line 57:

```
    @Composable
    fun containerElementFactory(component : List<Container.Element>?) {
        component?.forEach {
            when(it.__typename){
                "Typography" -> Typography(data = it.typography!!)
                "Button" -> GeneralButton(data = it.button!!)
                "Image" -> FallbackImage(url = it.image?.url!!, alt = it.image.alt)
                "Card" -> Card(card = it.card!!)
                "TextInput" -> TextFieldInput(textInput = it.textInput!!,
textFieldValue = textFieldValue )
                "Banner" -> Banner(banner = it.banner!!)
```

```
                "Chip" -> TextChip(chip = it.chip!!)
                "IconButton" -> IconButton(iconButton = it.iconButton!!)
                else -> Text(text = "Nothing to show ")
            }
        }
    }
}
```

Now, since our container element is in place, we now need to find a way to display our Containers based on what was returned by our API, and that's where our containerFactory Composable Function comes in, to do that place the code in line number 73 of the main Activity:

```
@Composable
private fun containerFactory(component : List<SampleScreenView.Element>) {
   Row (
       Modifier
           .fillMaxSize()
           .fillMaxWidth()
           .padding(8.dp)
           .verticalScroll(rememberScrollState())){
       Column {
           component.forEach {
               when(it.container!!.containerType){
                   ContainerType.COLUMN -> Column(modifier =
Modifier.padding(4.dp)) {
                       containerElementFactory(component = it.container.elements)
                   }
                   ContainerType.ROW -> Row(
                       Modifier
                           .horizontalScroll(rememberScrollState())
                           .padding(4.dp)) {
                       containerElementFactory(component = it.container.elements)
                   }
                   else -> Column {
                       containerElementFactory(component = it.container.elements)
                   }
               }
           }
       }
   }
}
```

With that in place now, let’s call our function. To do that just place this in line 50 in the MainActivity:

```
containerFactory(component = res)
```

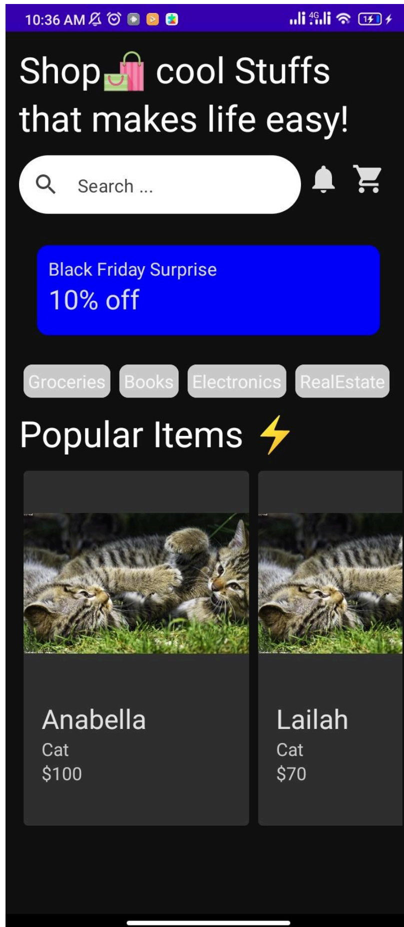
At the end of this our MainActivity should look like this:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.horizontalScroll
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.rememberScrollState
import androidx.compose.foundation.verticalScroll
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.TextFieldValue
import androidx.compose.ui.unit.dp
import androidx.lifecycle.lifecycleScope
import com.apollographql.apollo3.api.ApolloResponse
import com.example.GetSampleScreenElementViewQuery
import com.example.fragment.Container
import com.example.fragment.SampleScreenView
import com.example.sdugaphql.component.*
import com.example.theapp.ui.theme.TheAppTheme
import com.example.type.ContainerType

class MainActivity : ComponentActivity() {
   lateinit var textFieldValue: TextFieldValue
   var response: ApolloResponse<GetSampleScreenElementViewQuery.Data>? = null
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       lifecycleScope.launchWhenResumed {
           response =
apolloCleint.query(GetSampleScreenElementViewQuery()).execute()
           setContent {
               TheAppTheme {
                   // A surface container using the 'background' color from the
theme
                   var textFieldValue1 by remember {
                       mutableStateOf(TextFieldValue())
                   }
                   textFieldValue = textFieldValue1
```

```
                    val res by remember {
                        mutableStateOf(
response?.data?.sampleScreen?.sampleScreenView!!.elements
                        )
                    }

                    Surface(
                        modifier = Modifier.fillMaxSize(),
                        color = MaterialTheme.colors.background
                    ) {
                        containerFactory(component = res)
                    }
                }
            }
        }
    }
    @Composable
    fun containerElementFactory(component : List<Container.Element>?) {
        component?.forEach {
            when(it.__typename){
                "Typography" -> Typography(data = it.typography!!)
                "Button" -> GeneralButton(data = it.button!!)
                "Image" -> FallbackImage(url = it.image?.url!!, alt = it.image.alt)
                "Card" -> Card(card = it.card!!)
                "TextInput" -> TextFieldInput(textInput = it.textInput!!,
textFieldValue = textFieldValue )
                "Banner" -> Banner(banner = it.banner!!)
                "Chip" -> TextChip(chip = it.chip!!)
                "IconButton" -> IconButton(iconButton = it.iconButton!!)
                else -> Text(text = "Nothing to show ")
            }
        }
    }
    @Composable
    fun containerFactory(component : List<SampleScreenView.Element>) {
        Row (
            Modifier
                .fillMaxSize()
                .fillMaxWidth()
                .padding(8.dp)
                .verticalScroll(rememberScrollState())){
            Column {
                component.forEach {
                    when(it.container!!.containerType){
                        ContainerType.FILL -> Column {
                            containerElementFactory(component =
```

```
it.container.elements)
                       }
                       ContainerType.COLUMN -> Column(modifier =
Modifier.padding(4.dp)) {
                           containerElementFactory(component =
it.container.elements)
                       }
                       ContainerType.ROW -> Row(
                           Modifier
                               .horizontalScroll(rememberScrollState())
                               .padding(4.dp)) {
                           containerElementFactory(component =
it.container.elements)
                       }
                       else -> Column {
                           containerElementFactory(component =
it.container.elements)
                       }
                   }
               }
           }
       }
   }


}
```

Voila!! 🎉Our Server Driven UI App is ready and this is how our UI should look.



Note: Server-Driven UI isn't a silver bullet that solves all the issues that might be encountered with Client Driven UI. While this approach might look like a better candidate, you should ask yourself this question: will this really not make our codebase even more complex? what part of UI should be driven by the Server? I think you could find the answer to this questions. If you thought to use Server Driven UI, this will really help your team decide if it's the right decision.