Checkmate on Compose — Part I

Zsolt Kocsi · Follow
Published in Bumble Tech
13 min read · Aug 31, 2021

Listen Share

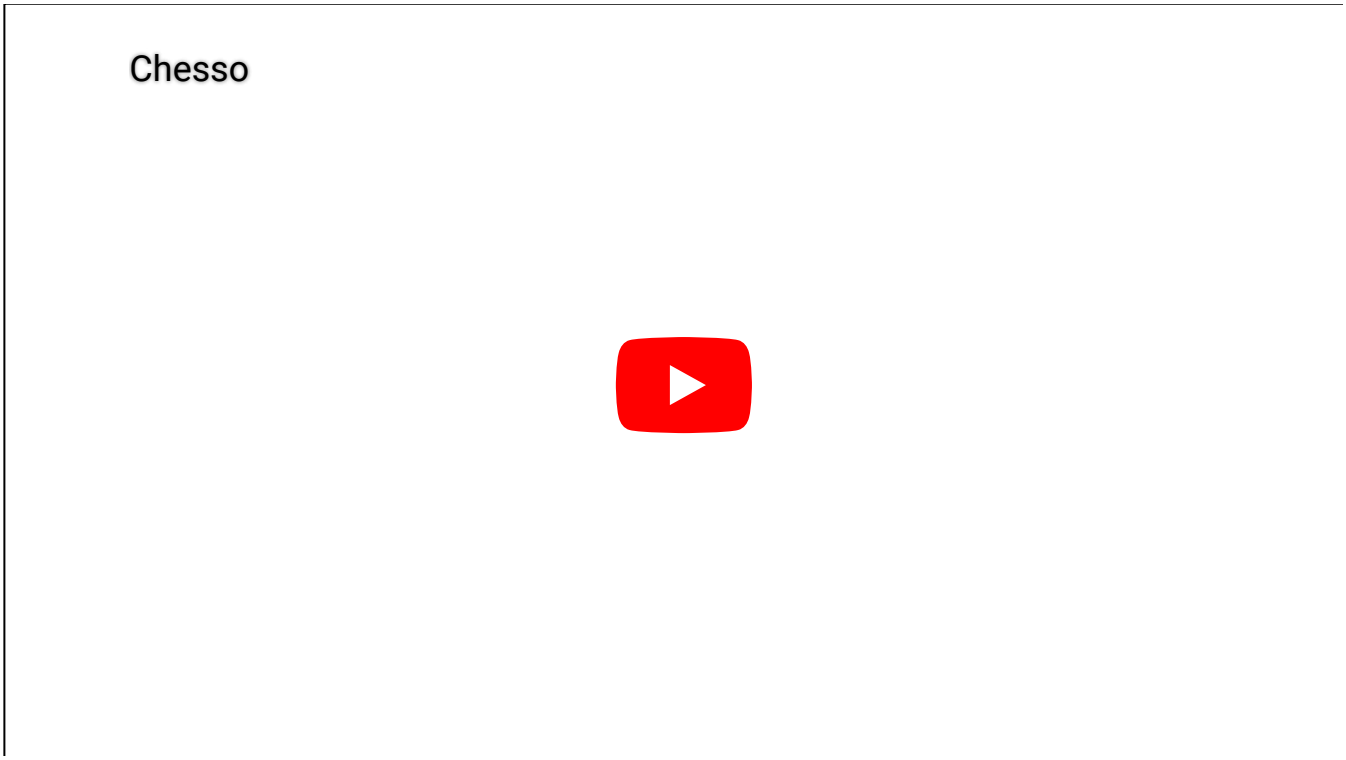
Lessons learned from a Jetpack Compose-based chess app

A sneak peek at what this two-part article series is about

Here at Bumble Inc— the parent company that operates Badoo and Bumble apps — we're in the middle of adopting Compose, so a lot of us play around doing pet projects to improve our skills and then we share some best practices.

I've created a chess app: https://github.com/zsoltk/chesso

Chesso is an animated, Jetpack Compose-based app aimed at beginners and intermediate players — its distinctive key feature is the ability to show visualisation layers on top of the board:



In this two-part article we explore the principles on which it was built:

- **Part I** (this article): fundamentals of the game and UI
- **Part II** (next article): animations and visualisations

Motivation

When I first started playing chess properly (after multiple previous half-hearted efforts) there were a few features I felt were really missing from the chess apps I found on the market.

For example, the chess engine might tell me that a particular move is good or bad, but not why. Sometimes the answer does require an understanding of more advanced concepts of the game, true, but the answer isn't always complicated.

> For a beginner, I believed that overlaying some visualisations would be a great help when tackling some simpler situations.

These might include for example:

- Pressure on a given square (from both sides)
- Blocked pieces

- Active pieces
- The king's direct (and indirect!) escape squares when trying to checkmate
- etc.

Also, in some chess apps when you accidentally stalemate your opponent instead of a checkmate, or blunder yourself into a checkmate, the game is immediately over and there is no option of going back and trying something different. This makes

I had followed the evolution of Jetpack Compose throughout its first 6 developer preview versions or so but hadn't had the time to keep up with all the changes that were constantly being introduced.

With a stable 1.0 already around the corner in the spring of 2021, a moderately complex app like Chesso was the perfect opportunity for me to get up-to-date with the latest APIs, and implement the above ideas.

After all, how hard could it be?

(Some months later I can say: it wasn't *that* hard — but, as is all too common with pet projects, it did take me longer than I'd imagined)

What we will (not) create

The goals in this project were to:

- Create a fully functional chessboard with Jetpack Compose
- Play around with animations in Compose
- Implement an extendable visualisation layers approach
- Allow importing games from other apps (to analyse them with the visualisations)
- Have some fun!

However, I did ***not*** want to:

- Create a fully functional chess app with chess engine integration and online multiplayer functionality — there are already great apps out there doing just that!
- Create an efficient game engine — you could definitely optimise algorithms of the game mechanics to a greater extent than I did.

Let's get started!

The chess parts — fundamentals

Before we can get to the UI and Jetpack Compose-related parts, we first need to implement some fundamentals of chess first. Fortunately, they aren't too complicated.

- **Set**: There are two sets of pieces: black and white — this can be an enum
- **Piece**: There are pieces. A piece is part of a set, has an asset and text symbol, and has a value that will come in handy with calculating score
- **Position**: positions of the board (using Algebraic Notation) can be easily modelled with a simple enum as well: a1, a2, ..., h8
- **Board**: we can now have a board. A board has an association between positions and pieces

Let's pause at this point for a moment.

> You might feel a strong temptation to reuse this same class for adding more and more complexity in the game.

But I recommend you resist the urge and stick to keeping things nicely separated into layers. Each layer wraps the previous one, adding something to it:

1. A **Board** is just that: pieces in positions. Like in real life, by merely looking at the board we couldn't infer much about the state of the game.

2. A **GameSnapshotState** adds more information about the game itself. It still only describes one particular state of the game, but in addition to visual information on the board, can also tell us about e.g.
   a) whose move is it next?
   b) resolution (still in progress or finished?)
   c) some more nuances (can the white/black king still castle queenside/kingside with this state, or has that option been lost?)

3. **GameState** then holds all the information about the entirety of the game: a list of multiple individual **GameSnapshotStates**, an index (to denote which one of the states we are currently displaying) and some meta info (e.g. player names, date, event, etc.).

4. **GamePlayState** adds state that's unrelated to chess but related to the app itself: like the state of the UI (e.g. if there's a piece selected) or if there are any active visualisations selected.

This approach also allows us to implement **time travel** for the game easily. We can change the displayed state index in **GameState** to go back any number of moves, then choose to go ahead with a different move by discarding the remaining snapshot states and adding a new one there.

The chess parts — game mechanics

It's beyond the scope of this article to detail all the game mechanics. However, apart from a few tricky bits, most of them are pretty straightforward, relying on state, positions, and easy-to-formulate rules.

Pseudo-legal moves and check

For calculating legal moves and captures, it's best to first create a list of pseudo-legal moves:

1. All the moves you could make with a given piece based only on its movement rules alone.

Then filter this list by applying check constraints:

2. Any move which would result in a game state having check against the current player is filtered out.

This simple rule covers all different cases nicely:

- If your king is in check, you can only make moves that get you out of check.
- If a piece is pinned, the rule automatically limits you from moving it out of the line of attack against the king.
- Nor can you move your king into positions of check.

Game resolution

Following on, it's now very easy to determine **checkmate** and **stalemate**:

1. Let's take all the pseudo-legal moves.
2. Let's filter them by applying the above check constraints.
3. Did we end up with an empty list? No? Carry on with the game.
4. If there are no legal moves available: is the king in check? If so, it's checkmate — otherwise, it's a stalemate.

Draw by repetition can be also easily implemented with our established data structures:

1. Take the list of game snapshot states (Remember, it includes not only the board state but e.g. castling availability, which is important for this rule)
2. Map them to their unique hashcodes (We don't even need to evaluate the board and the state ourselves! A very convenient shortcut.)
3. See if there are 3 of the same hashcode value present in the list.

And that's it!

Composing the board

Finally! Now that we have the mechanics in place, let's get down to the UI parts.

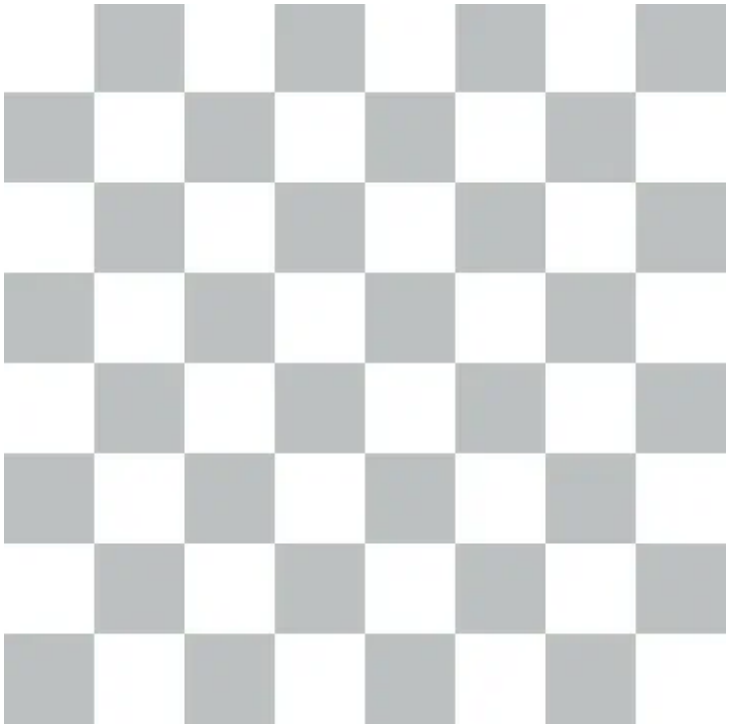
We'll want to:

1. Paint squares
2. Render pieces
3. Add some decorations like highlight or legal moves
4. Render moves
5. Animate the board
6. Add visualisation layers

Painting a square

This one shouldn't be too difficult! You can either go about it implicitly (use e.g. a *Box* composable to nest other layout elements, and just add a background colour) or explicitly (use the *Canvas* composable and draw a rectangle). The difference is for the most part purely semantic here.

Painting 8×8 squares to fit the screen



8×8 empty board

We'll want our chessboard to:

1. Have a 1:1 aspect ratio
2. Fill the available screen space

A first approach could be:

This layout gives us:

- A square box filling the max available size
- → containing 8 equally-weighted rows packed vertically
- → each of them containing 8 equally-weighted squares packed horizontally

This can work initially, however, in the end, I needed to take a very different approach. (We'll return to this later when we discuss animation).

Rendering a piece

Fun fact: did you know that chess pieces are part of Unicode? All of these are in fact text symbols:

♔♕♖♗♘♙♚♕♜♝♞♟

For simplicity I went with these originally and just rendered pieces as text:

However, it proved not to be the best route in the long run:

- The size is defined as font size, which can be inconvenient for graphics
- The text symbols for light pieces aren't actually filled: they're only a thin black contour with transparency in the middle and so when rendered against dark coloured squares they don't look that great

Text-based white pieces don’t look that good on dark squares

So, in the end, I opted to render pieces as vector assets instead:

Board with vector pieces

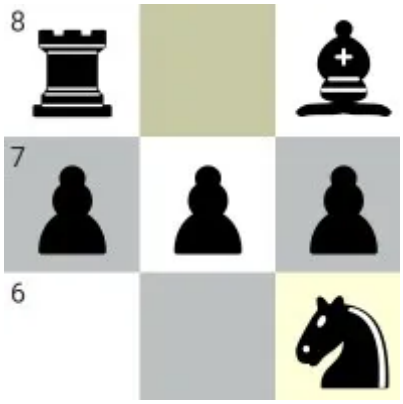
> Note the passed *tint* parameter to the *Icon* composable. Without specifying this, Compose will apply a default tinting over your assets.

On the plus side, text symbols are still great for:

- Showing a list of captured pieces
- Showing moves text with a figurine notation (e.g. "♘f3" vs "Nf3")
- Or if you want to create a purely text-based chess app that you can play in a terminal, for example

Adding visual clues

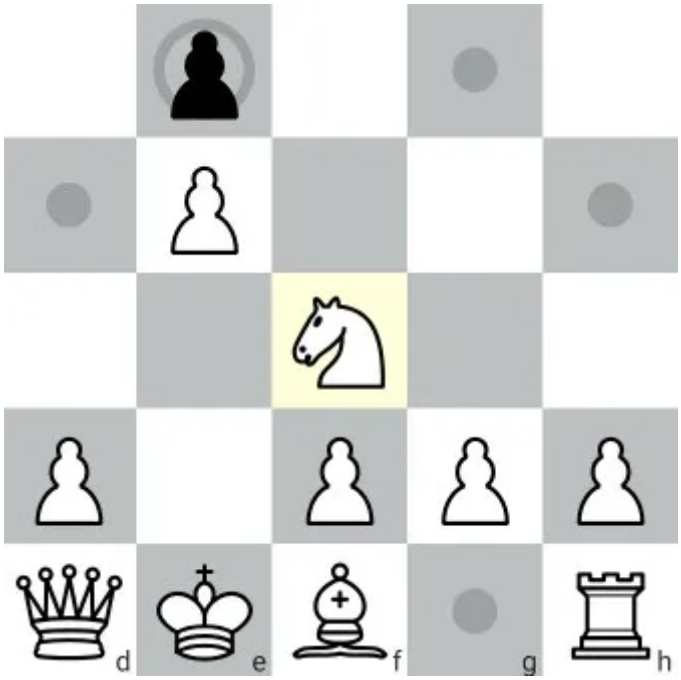
Now, let's add some visual clues! First, highlight the squares the last piece moved from and to:



Nc6

This can be done in the same way as painting the square background itself with an extra alpha value for transparency.

Next, add some target marks on squares the currently selected piece can move to (if it's an empty square) or capture at (if it has the opponent's piece):

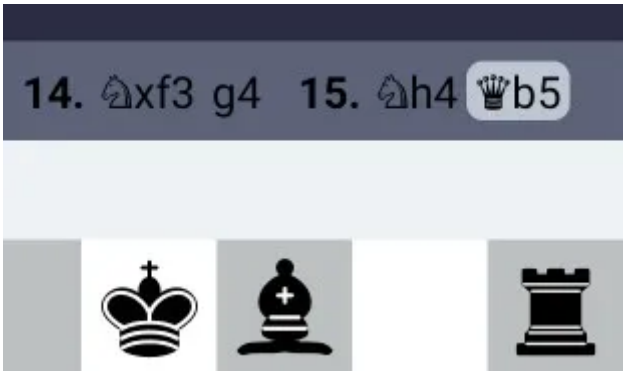


Knight's legal moves from f3

I did both with the same composable using slightly different parameters:

Creating the moves list

Any decent chess app needs to show not only the current state of the board but also a list of moves made so far:



List of moves with figurine algebraic notation

To implement this, we'll need:

- a horizontally-scrollable list
- a pill shape
- to conditionally add this pill shape for the selected move

Nothing too complicated, fortunately. Here's the list part:

Some notes:

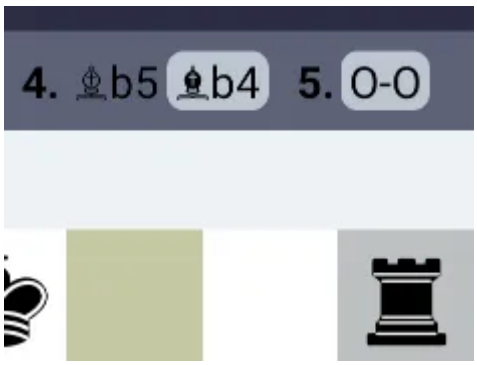
- What I considered to be a move actually only count as a ply (or half-move) in chess — we should only show the incremented move number once after every 2

half-moves.

- The key passed to an item is an optional parameter for Compose, but it plays an essential role here since it contains info not only on the index but on the state of selection too — and in the above snippet, we render selected moves with a pill background.

> Keys are expected to be stable and unique. Omitting the selected boolean parameter here would hide the fact that the "same" item needs to be rendered differently based on its value, leading to weird UI bugs.

Like this one, where it looks as if multiple game states are active at the same time:



UI bug: multiple pills in the move list

So don't forget: while it's easy to think of them as the "same" items, any and all additional variables that can affect their looks will make them **different** items, and so they should be included in the key as such!

> Musing: I wish Compose also offered an *animateScrollToKey* operation rather than the *animateScrollToItem* (which expects a numerical index).

The pill shape isn't too difficult. You can achieve it by simply passing a *background Modifier* to a *Box*, defining a *RoundedCornerShape*. Then allow this to render any generic composable passed to it to make it reusable:

The power of modifiers

Modifiers are an extremely powerful concept in Compose. They allow a wide variety of functionality to be defined from looks (background, padding, size, etc.) to behaviour (e.g. scroll/swipe / drag/click functionality).

> *For a nice overview of Modifiers, I suggest visiting: https://foso.github.io/Jetpack-Compose-Playground/general/modifier/*

What makes them ultimately powerful though, are the following:

1. Modifiers are object instances that we can pass around the code

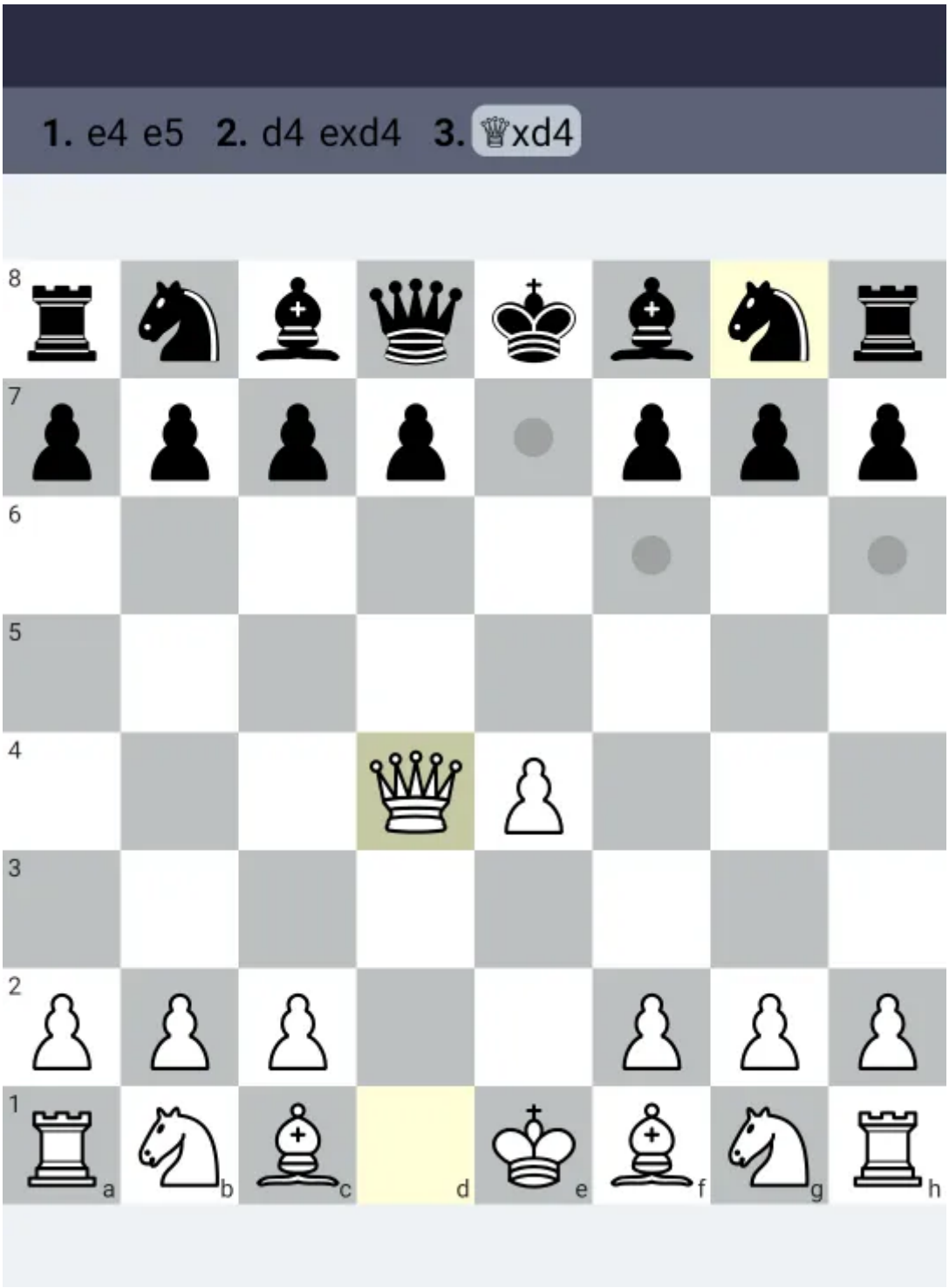
2. Modifiers can be chained, so we can combine them further (e.g. receive an instance in a composable and add something to it)
3. Since virtually all composables accept an optional instance of a *Modifier*, we can share custom looks and behaviours across otherwise widely different parts of our app

For now, we'll just do something simple now and convert our **Pill** composable to a *Modifier.* To be precise, it will be an extension method over any Modifier instance, so we can chain it:

Then instead of wrapping our *Move* composable into another one, we can just use this *Modifier*. Now we can easily give anything a pill background!

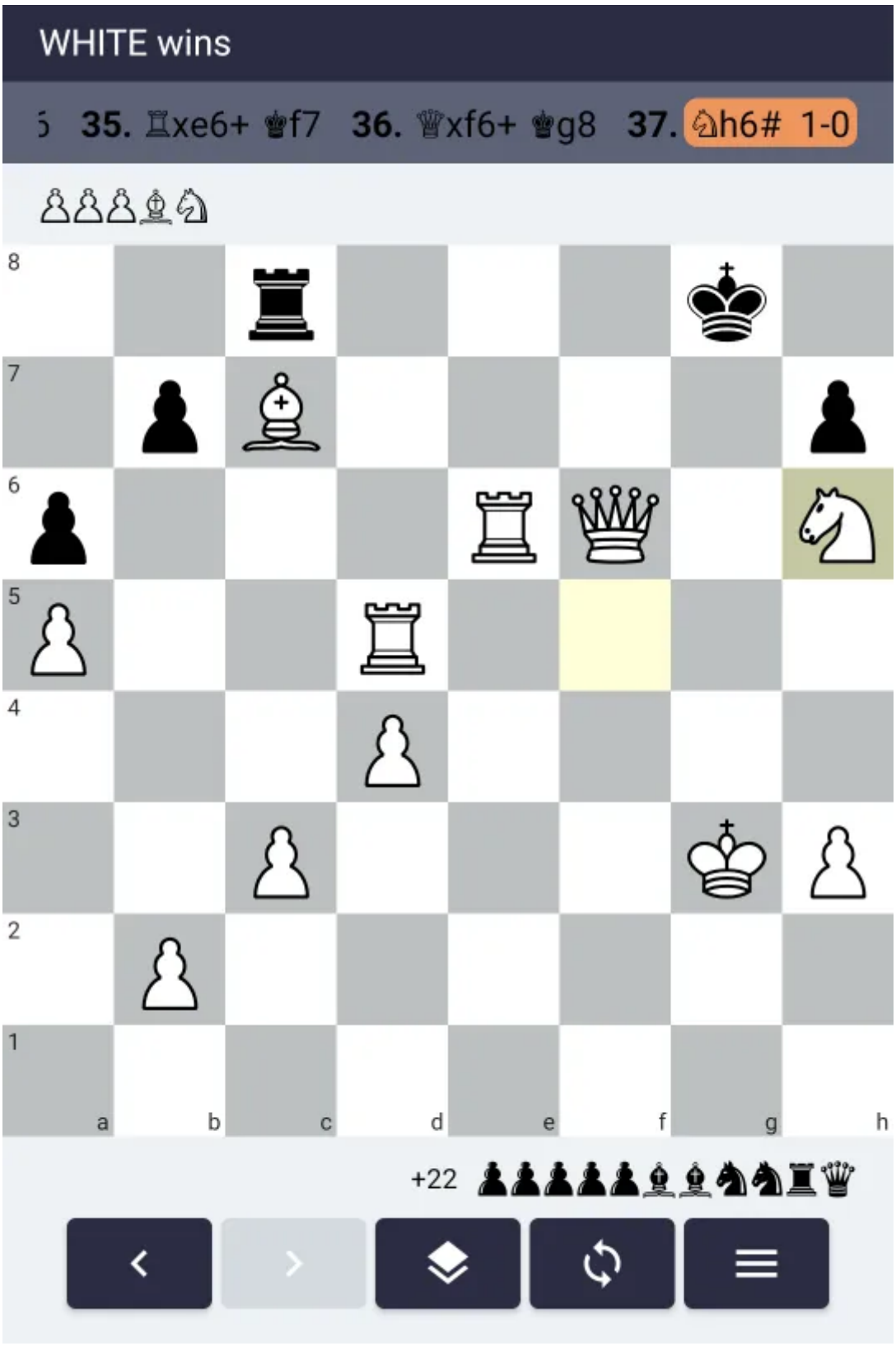
What we have at this point

We've created a chessboard with pieces, highlights, target marks, and a moves list:



Not too bad!

Add a few more extras and we'll have a proper chess app:



Added here are:

- A simple status text on the top of the game's progression
- A differently coloured pill for moves with check and checkmate
- A list of captured pieces

- Some buttons to move back and forth in the game, show visualisations, flip the board and trigger a game menu

As most of these elements are very simple in themselves (texts and buttons) I won't detail them here, but you can always delve into the source code if you're interested.

For reference, here is our rough Composable hierarchy at this point (with details and data models stripped from it):

General takeaways

Logic and UI state

I initially found myself passing down game state (the business logic state in general) through composables. The more complex the app got, the more painful this approach became.

I refactored the code so that composables receive UI state rather than business logic state, and I highly recommend you do the same.

Code not only became simpler, but this way we can also keep the Single Responsibility Principle: our composable will only have to deal with what to render, and not how to decide what to render.

State & Reducer

I resisted the urge to use my go-to approaches as a default (we use MVICore at Bumble) and started exploring what I could achieve in Compose without them. It does, after all, handle state on its own, so why bother?

> And indeed, half the code we usually put into Reducers are better handled by Compose: if the state is close to the UI, it's simply a lot more convenient to work with it and avoid jumping through any additional hoops.

However, as the game grew more complex, I still found that game state-related changes were easier to handle in a Reducer.

Maybe it's just because chess and other board games are inherently easy to represent with distinct states and actions over those states. But regardless of whether you want to go with a Reducer or not, it's fine to have your business logic outside of the realm of Compose.

MutableState APIs

The app hosts a total of 4 dialogs so far:

1. Promotion dialog (when a pawn reaches the last rank and you can pick what piece to promote it to)
2. Active visualisation picker dialog

3. Game dialog (new/import/export)
4. Import dialog (imports a game from a PGN-encoded text)

All of these dialogs are state-dependent:

> In Compose you don't just "launch" a dialog and forget about it (old reflexes!), you declare that there's a dialog based on your state — and you also need to clear this state.

One way of doing it is to pass some callback to your dialog itself, which would trigger it in *onDismissRequest.*

Compose is unopinionated which API you want to go about handling state— see this excerpt from https://developer.android.com/jetpack/compose/state:

> *There are three ways to declare a* `MutableState` *object in a composable:*
>
> ```
> val mutableState = remember { mutableStateOf(default) }
> var value by remember { mutableStateOf(default) }
> val (value, setValue) = remember { mutableStateOf(default) }
> ```
>
> *These declarations are equivalent, and are provided as syntax sugar for different uses of state. You should pick the one that produces the easiest-to-read code in the composable you're writing.*

1. The first option returns a *MutableState* type.
2. The second option uses delegation, and won't work for writing the state outside of the scope of the current method that it's declared in.
3. The third option is nice and clear, and looks like React hooks.

However, for N dialogs that live in the top-level scope of the game, this will give us N * 2 parameters (get & set dialog state), which can blow up the code pretty fast, especially when both the getters and the setters need to be passed to deeper layers.

So in the end I went with the first option, even though it felt weird to pass a bunch of *MutableState* around. On the plus side, it makes it very explicit when looking at the code that receives a *MutableState*, that you will want to change something in a higher scope.

The fancy bits are coming — To be continued

So far we've covered the foundations of a chess app and static pieces of the UI using Compose. We could call it a day and be happy with it, however, there are more matters to explore:

- How do we animate pieces between moves?
- How do we add the signature visualisations I talked about in the beginning?

We'll cover these in the follow-up article: Checkmate on Compose — Part II.

In the meantime, come over to https://github.com/zsoltk/chesso — contributions welcome!

Android Android App Development Programming Jetpack Compose

Mobile App Development

Follow

Written by Zsolt Kocsi

450 Followers · Writer for Bumble Tech

Principal Android Engineer @ Bumble