For the implementation of a new feature within the client, several aspects must first be considered. Nearly every feature is broken down into three components: the feature view, the feature model, and the interfaces required to generate the content of the feature.

During our development, we would typically begin a feature’s implementation with the feature view. By beginning a feature’s development with the feature view, we were able to prioritize the user-experience and develop a clearer goal of our features based on what would be most intuitive as a user. The feature view essentially consists of a widget representing a page within the app, along with any sub-widgets representing individual graphical components within the page. Our top-level page widgets generally started off as stateless widgets, with certain widgets being refactored into stateful widgets as their requirements and responsibilities became better defined. This allowed us to minimize each widget’s internal state management as much as possible, instead delegating state management to our feature models.

After developing a general prototype of our feature’s view, we would move onto implementing the feature model. While our stateful widgets manage small sets of internal state data, a more accessible and persistent state model was necessary for our service. To address this, we utilized Flutter’s scoped models. Nearly every key feature possesses a feature model which contains important state data relevant to that feature. These models also provide an interface for managing the feature states from anywhere within the app. Once a feature model is created, it is included as a reference within our app-wide model which is passed into the widget hierarchy at the root of our application.

After creating both the feature view and model, we would move onto considering how the feature would interface with our backend or third-party services. After determining which backend lambda functions would be relevant to a specific feature, the respective API requests would be written as modular functions within an API service class. Some features required additional Flutter plugins to generate their content as well, which would be imported and utilized within a wrapper class.

After implementing our feature services, we would then integrate the appropriate calls within the feature model. We would then add our model references within the feature view and call the various model functions for accessing and managing the feature’s state data.

One of our main considerations in implementing new features was ensuring a responsive user-experience. Many of our features rely on retrieving and updating data from our backend, which led to challenges in rendering our views quickly. Our solution to this was utilizing Flutter’s future builder widget, which allowed us to display placeholders while our API requests were sent and the responses were received and processed.