Universal App Browser - это путь в будущее.

Для меня идеальный GUI это app, который не требует затрат на программирование, дизайн, обслуживание и способный одинаково работать с любыми языками, и на любой платформе без всяких подстроек. Возможно ли это при нашей жизни мы попробуем разобраться.

Несложно освоить по отдельности что-то из Vue/React, JavaFX, Python PyQt, .. но получать данные и взаимодействовать с зоопарком софта простым и элегантным способом, не думая о user-OS/броузерах/платформах - нерешаемая задача для подобных инструментов. Я не хочу лезть в новый фреймворк (даже в старый, забытый), меняя язык программирования, выгребая грабли и забивая голову мусором. Хочу программить именно мою задачу, не отвлекаясь на борьбу со всякими GUI фреймворками. И для себя нашел решение.

В качестве протокола обмена будем использовать Json как топовый формат по условиям популярность/понятность/читаемость/поддержка у всех программных языков в той или иной мере.

Сервер отдает Json данные, по которым наш app GUI должен задизайнить картинку, удовлетворяющую спекам красивого GUI. Google со своим Material Design на сегодня является стандартом, поэтому берем его.

Требования к современному GUI включают наличие стандартных элементов, таких как кнопки, поля ввода, таблицы и т. Д. Прикинем, как используя минимальный набор соглашений сказать GUI, что нам нужны некие элементы на экране.

Кнопка без состояния {‘name’: ‘Push me’}. Если элемент содержит только имя то он кнопка.

Поле ввода {‘name’: ‘Edit me’, ‘value’: ‘’} потому что тип value - строка.

Кнопка свитчер {‘name’: ’My state’, ‘value’: false} потому что тип false - boolean.

Выбор из списка {‘name’: ‘Switch something’, value: ‘choice1’, options=[‘choice1’, ‘choice2’,’choice3’]}

Картинка {‘name’:’Image, ‘image’ : ’some url’, ‘width’: .., ‘height’ : .. }

Таблица {‘name’:’My table’, headers=[‘Name’, ‘Synonym’], rows = [

    [‘young’, ‘youthful’],

    [‘small’, ‘ meager’],

    ...

]}

Для кастомизируемости можно задать тип элемента {type: ‘Switcher’}, если автоматически подобранный по JSON тип не устраивает. Такое возможно, когда на одинаковый JSON может быть отображен более чем одним способом. Например ‘Выбор из списка’ может быть представлен как поле ввода с выпадающим списком, а может и как горизонтальный набор кнопок, одна из которых активна и соответствует текущему value. При малом количестве опций имеет смысл автоматом использовать кнопочный вариант, при большом (более 3) - поле ввода-выбора. Наш GUI сам выбирает оптимальным для отрисовки способом, но если сильно нужно - type : .. в помощь. В норме тип не нужен, автодизайнер справляется сам.

Дополним картину по мелочи:

- если имя не должно отображаться на экране, оно должно начинаться с _;

- если нужно где-то подрисовать иконку, добавляем ‘icon’: ‘любое имя стандартной Material Design иконки’; push кнопка-иконка соответственно будет описана как

{‘name’: ‘_Check’, ‘icon’: ‘check’}

- сложные элементы управления такие Таблицы, Viewers, могут иметь дополнительные параметры, ‘editable’, ‘scroll’,.. которые могут использоваться для специфической отрисовки или выбора доступных опций редактирования/поиска/выделений для таблицы. По умолчанию - unigui сам определяет функционал, анализируя какие обработчики назначены таблице, графику или строке редактирования.

Для логической группировки элементов введем понятие блока, который группирует логически связанные элементы в один визуальный блок.

{name: ‘Block 1’, ‘elems’ : [{‘name’: ‘_Check’, ‘icon’: ‘check’}, …]}

В пределах блока все элементы должны иметь уникальные имена, ибо они же id.

В норме блоки строятся горизонтально, если они не могут все влезть на экран (запустили app GUI на мобиле например), автодизайнер скрывает их, но добавляет в тулбар иконки, позволяющие открыть их тапом. Это дает опцию работы со сложным GUI даже на маленьких экранах. Блоки могут конфигурироваться и произвольно, подробнее по ссылке в репозитарии в конце.

Верхний уровень описания - Screen. Имеет вид

{‘name’: ‘Screan’, blocks: [..], menu : [{‘name’: ‘Screen’,’icon’: .., }, ..],

‘toolbar’: [набор JSON - элементов (кнопки, поля, что угодно)]}

Генерируется автоматически.

uniGUI - отдельный процесс, связывающийся с сервером данных по Websocket и обеспечивающий отображение его данных с последующим информированием обо всех важных для сервера обновлений этих данных.

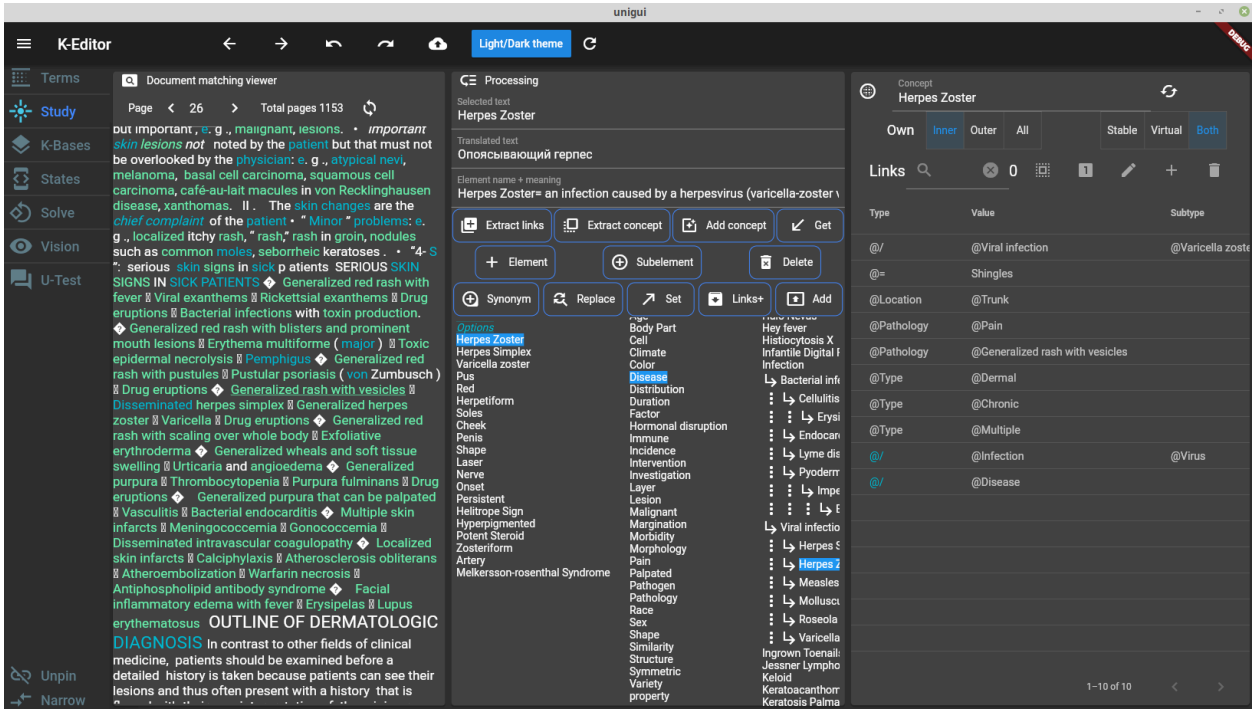
При коннекте к серверу uniGUI ожидает получить Screen. Получив его он на лету рассчитывает оптимальный дизайн и рисует полученную инфу оптимальным для экрана юзера образом, дальше ждет реакции от юзера и сервера.

От построенного изображения данных сервер получает поток сообщений JSON, которые полностью описывают что сделал юзер. Имеют вид [‘Block’,’Elem’,’type of action’,’value’], где ‘Block’ и ‘Elem’ - имена блока и элемента, value - значение события. Сервер может либо принять изменение либо откатить их, послав инфо окно о несоответствии. Может поднять диалоговое окно, которое описывается как блок, и имеет доп. параметр ‘buttons’, который описывает кнопки диалога. Клиент мгновенно отображает актуальные данные сервера и их изменения. Пересылаются только измененные сервером объекты.

Получать события и обеспечить их обработку сделаем Websocket прослойку (фреймворк), которая будет автоматом транслировать сообщения в вызовы обработчиков, которые привязаны к нашим данным (объектам).

Задача обеспечить автоматическую трансляцию в JSON, следуя формату, занимающему две страницы A4, вполне решаема на любом языке. Надеюсь, что так.

Чтобы не обсуждать применимость этого подхода для сложных app-ов , ниже привожу скрины такого как описано uniGUI, написанного на flutter. Выбор упал на него за многоплатформенность и отсутствие дополнительных слоев типа JS/html/chrome. В минус flutter можно записать плохую поддержку десктопа и низкое качество кода верхнего слоя (GUI элементы), неудобную архитектуру для точечных обновлений и управления элементами как данными, что, впрочем, лечится.



Вся магия на сервере сводится к тому, что наши отображаемые данные должны быть сцеплены с прослойкой таким образом, чтобы без всякого кодинга обеспечить их автоматическую трансляцию в JSON и вызов обратных уведомлений от активности юзера. Так это зависит от возможностей конкретного языка, то под каждый язык прослойка может иметь разные варианты как архитектурно, так и конкретно.

Например, в одном случае прослойка имеет папку скринов, каждый из модулей в котором имеет описание одного экрана на Python. При старте прослойка считывает экраны, при начальном коннекте отдает юзеру тот, у которого глобальная priority = 0. Все данные автоматически транслируются и декодируются с помощью jsonpickle и json.loads. Полное описание блока, который крайний справа на скриншоте вверху, обеспечивающее автоматическое связывание и отображения данных программы в uniGUI и передачу событий обратно выглядит так:

```
Links = Table('Links', headers=links_headers, rows=[],   value=-1)
```

```
concept_name = Edit('Concept', value='')

own_links_select=             Select('Own',             value=own_types[0],
options=own_types,changed=change_type_links)
ext_links_select       =       Select('_Status',       value=ext_types[2],
options=ext_types, changed=change_type_links)
details     =     Block('_Details',[concept_name,     Button('_Name_update',
icon='cached',changed=update_name)],
        [own_links_select,ext_links_select ],links, icon='blur_circular')
```

Поток сообщений app GUI -> сервер выглядит так:

flutter: [Glossary, Terms, =, 658]
flutter: [_Details, Links, @, @Folliculitis]
flutter: [_Details, Links, @, @Adolescent]
flutter: [toolbar, _Back, =, _Back]
flutter: [toolbar, _Forward, =, _Forward]
flutter: [toolbar, _Back, =, _Back]
flutter: [_Details, _Status, =, Virtual]
flutter: [_Details, _Status, =, Stable]
flutter: [_Details, Links, @, @Inflammation]
..

А не будет ли такой GUI тормозить при ожидании событий с сервера. Нет, потому что GUI работает в режиме уведомлений сервера, реагируя на все действия юзера как нормальный локальный GUI. По умолчанию он считает молчание сервера на действия юзера как Ok. При смене экранов понятно, что должно прийти его описание. Тут возможна задержка по сравнению с типичным app с embedded GUI.

Итог. Возможно уже сегодня, на текущих инструментах сделать GUI, убирающий >95% стандартного, нудного GUI кода обслуживания и не беспокоится как и на каком устройстве работать. По крайней мере, для бизнес/сайенс приложений. Эта статья - Proof of concept того, что создание универсального App Browser не только возможно, но и необходимо.

P. S. Эта статья некоторое время назад была опубликована на популярном ресурсе русскоговорящих программистов (habr) и мнения относительно ценности этого концепта разделились. Примерно половина увидела просто новый фреймворк с фичей JSON-разметки вместо XML, сочтя универсальный GUI протокол и избавление от программирования фронт-GUI вещами незначительными или сомнительными. Поэтому здесь я добавлю, что именно в будущем даст подобная технология. Наверное многие могут вспомнить фантастические фильмы, где показано как круто главный герой серфит в

VR по инфо-сайто-ресурсам, пальцами вытаскивая нужные блоки визуализированной информации и комбинируя их с другими блоками, на лету производя трансформации над ними и тут же отправляя их на третьи ресурсы. Что-то похожее пытались изобразить в фильмах Джонни-мнемоник, Особое мнение и т.д. Для того чтобы подобное стало возможным, нужны инфоресурсы, способные отдавать информацию в некотором стандартном виде, способном мгновенно отображаться у пользователя в виде понятного GUI и чтобы данные, которые стоят за этим GUI были совершенно прозрачны и доступны для пользовательского app browser. Это в свою очередь позволит работать с любым множеством таких инфоресурсов как массивом однородных данных, в рамках одного GUI пространства пользователя, проводить любые доступные операции, в том числе перекрестные, на лету, как в фантастических фильмах, только на самом деле.

Repo https://github.com/Claus1/unigui