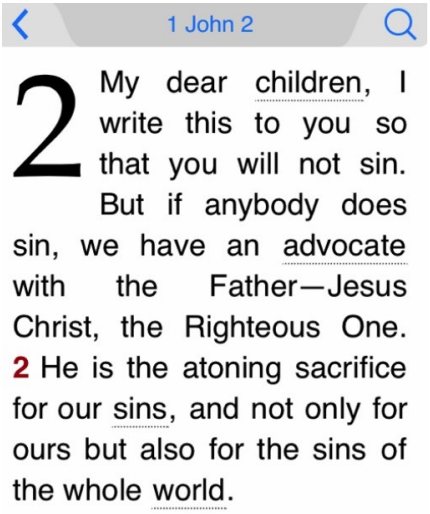
Key It Design Documentation

Table of Contents

Key It User Interface Basics

Start From Printed Scripture



Consider this tiny fragment of I John 2.

Consider someone keyboarding this fragment to be put into Adapt It (whether AIM or AID). It is reasonable to assume the keyboarder has a small basic knowledge of Bible text. This would include:

- recognising which Bible book it is
- recognising the beginning of chapters
- recognising the beginning of verses

For this fragment, that is all we assume. Later parts of this design will consider other aspects of Bible text and will add a few further assumptions about the knowledge of the keyboarder.

We would want this fragment to be exported from Key It into a text file like this:-

```
\id 1JN 1 John Tok Pisin, fragment chapter 2:1 - 2:2
\c 2
\v 1 My dear children, I write this to you so that you will not sin. But if anybody does sin, we have an advocate with the Father – Jesus Christ, the Righteous One.
\v 2 He is the atoning sacrifice for our sins, and not only for ours but also for the sins of the whole world.
```

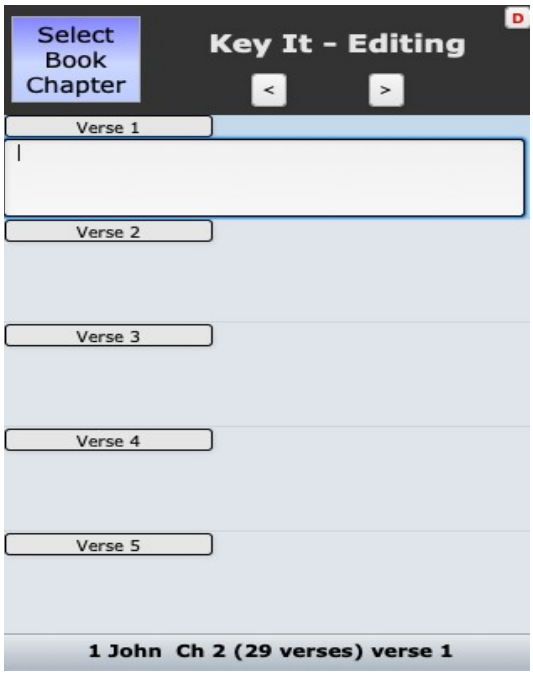
The task of Key It is to assist the keyboarder to type the text read from the printed page and to produce this output without requiring the keyboarder to know anything about USFM codes – not even their existence.

A stage in thinking through a design to achieve this is to write above each portion of the desired output a description of the type of that portion:

```
<Identification>
\id 1JN 1 John Tok Pisin, fragment chapter 2:1 - 2:2

<ChapterStart>
\c 2

<VerseStart>
\v 1 My dear children, I write this to you so that you will not sin. But if anybody does sin, we have an advocate with the Father – Jesus Christ, the Righteous One.

<VerseStart>
\v 2 He is the atoning sacrifice for our sins, and not only for ours but also for the sins of the whole world.
```

Allowing the keyboarder to select a book and a chapter would allow Key It to export the first two pieces of this fragment without requiring the keyboarder to know anything about \id and \c markers.

Presenting the keyboarder with empty cells, one per verse in the selected chapter, and then letting the keyboarder type the verse text into the cells would allow Key It to export the other two pieces of this fragment without requiring the keyboarder to know anything about \v markers.



The user interface that would be presented to the user would be something like this.

When the user is ready to export the chapter ready for AIM or Paratext, KIT would insert the correct USFM markers during the export operation. This design of Key It opens up the possibility of providing export to other forms as well, for example, USX or even RTF, because the logic for the export operation is separate from the user interface for keyboarding.

Items - not just Verses

By presenting to the user the correct number of empty verses in the chapter and book he has selected, and by not allowing the user to create or delete and verses hierarchy.

But we need to allow for publication matters that the user will see in the printed Scripture, and so will want to include in the Scripture text that he keyboards. These matters include:

- book titles,
- book introductions,
- subject headings,
- paragraph breaks,
- bridging of verses.

These can be handled by expanding the concept of "verses" to include all these other matters as well; so let's change the hierarchy to **Books owning Chapters owning Items**, and allowing Items to be of a number of types. Thus we would allow Item types like the following:

Verse	\v	Bible verse - cannot be created or deleted (except under strict conditions, see below)
Subject	\s	Subject Heading
Paragraph	\p	Paragraph Start
Title	\mt	Book Title
InTitle	\imt	Introduction main title
InSubj	\is	Introduction subheading
InPara	\ip	Introduction paragraph

The books, chapters and verses supplied in KIT would not include any of the publication matters like Subject, Paragraph, Title, etc. But KIT would allow the user to create or delete these as needed to match the printed Scripture being keyboarded from.

The formatting of items displayed as part of a chapter would be varied according to the type of the item in order to present an appearance to the Key It user that bore some resemblance to printed Scripture. But there is no need to make it an exact likeness.

Sorting Items For Display

Some creative thinking is needed to invent a way of sorting the Items so that the publication Items are displayed in their correct positions between the verse Items. Simply sorting by Verse number is not enough.

The scheme being trialled at present is as follows:-

- Each Item record has the VerseNumber that it is associated with.
- Each Item record is given an additional field called ItemOrder. In the FileMaker prototype this is a calculation field; SQLite or MySQL may use a number field into which appropriate order numbers are inserted by the functions that create the publication Item records.
- The value of the ItemOrder field depends on the type of the Item; this is held in the ItemType field.
- Calculations of ItemOrder are listed in this table:

InTitle	100 * VerseNumber - 90	Intro Matter Title (only before verse 1 of chapter 1)	\imt
InSubj	100 * VerseNumber - 90 + IntSeq	Intro Matter Subject Heading (only before verse 1 of chapter 1)	\is
InPara	100 * VerseNumber - 90 + IntSeq	Intro Matter Paragraph (only before verse 1 of chapter 1)	\ip
Title	100 * VerseNumber - 30	Main Title (only before verse 1 of chapter 1)	\mt
Chapter Description	100 * VerseNumber - 27	Chapter Heading (only before verse 1 of a chapter)	\cd
Chapter Label	100 * VerseNumber - 26	Label at start of a chapter (only before verse 1 of the chapter)	\cl
Ascription	100 * VerseNumber - 25	Ascription (only before verse 1 of a Psalm)	\d
Heading	100 * VerseNumber - 20	Subject Heading	\s
ParlRef	100 * VerseNumber - 15	Parallel Reference	\r
Para	100 * VerseNumber - 10	Paragraph start before a Verse	\p
Heading After Para	100 * VerseNumber - 5	Subject Heading after a paragraph break	\s
Verse	100 * VerseNumber	These records are supplied built into Key It; users cannot directly create or destroy these	\v
ParaCont	100 * VerseNumber + 10	Paragraph start within a Verse	\p
VerseCont	100 * VerseNumber + 20	Continuation of Verse that contains a paragraph break	

With the above formulae applied to Items, selecting the Items that have a chosen BookID and chosen ChapterID, and then sorting them in ascending order of ItemOrder places the Items correctly with the publication items interspersed between the Scripture verse Items.

The FileMaker prototype can display these ItemTypes and ItemOrders as a debugging/tutorial aid - just click on the small 'D' button at the top right of the user interface for chapter editing to toggle these on or off.

When looking at the FileMaker prototype, note that the labelling of each Item is done in terminology that the user will understand. For example, both Para and ParaCont are labelled for the user as Paragraph, because the user is not concerned with Key It's internal need to distinguish between a paragraph starting before a verse and a paragraph starting inside a verse. But Verse and VerseCont are labelled differently for the user because showing the user two "Verse 5" records would be confusing.

Controls for the User

Smartphones have very limited screen space compared to desktop or laptop computers. Therefor we should not consume screen space with buttons for creating or deleting paragraphs, etc. Instead the label telling the user the type of the Item also functions as a button to display a popover panel containing the necessary buttons for that particular Item.

Furthermore, the popover panel should contain only those buttons that are relevant to that particular Item. For example, the popover panel for a Verse Item must not contain a delete button - this is better than providing a Delete button and then giving an alert saying "you can't do that". In the case of Verse Items, something that could be there instead of a Delete button is a button to bridge that verse with the following verse.

Each type of Item needs to be considered and a list of allowable actions made so that the popovers can be programmed to show only those buttons relevant to that particular item.

It should be noted that most existing apps for smartphones and tablets have this sort of flexible interface and thus that Key It users are likely to expect this type of interface. Those of us who got into computing in the desktop era are more used to seeing arrays of buttons or menu items, only some of which are relevant at any particular time, but we are not designing Key It for ourselves!

Item Types and Their Allowable Actions

Initial Designs Using Small Numbers of Buttons

It would be nice if we could limit the actions relevant to each type of Item to just 4 actions, so that we could display a 2 by 2 grid of buttons in each popover. The following table is being written with this aim in mind; if we find that some items need more than 4 possible actions, so be it, but it would be nice if we could keep the popovers simple.

With this aim in mind the columns of actions are not labelled 1 to 4; they are labelled
TL (top left) TR (top right) BL (bottom left) BR (bottom right).

The aim is to put actions doing some sort of "creating" on the left and actions doing some sort of "deleting" on the right. Whether it works out this way remains to be seen!

Item Type	TL	TR	BL	BR
Title	nil	insert Heading after	delete Title	nil
Heading	nil	insert Para after	delete Heading	nil
Para	insert Heading before	nil	delete Para	nil
Verse	insert Para before	insert ParaCont after	nil	bridge with next verse
ParaCont	nil	nil	delete ParaCont	nil
VerseCont	nil	nil	delete ParaCont	nil

The above table was modified after matters such as how to represent bridged verses were decided. It turned out that we could design a single row of three buttons with one or two "insert" buttons on the left and just one "delete" button on the right.

This is the design that was implemented in the FileMaker prototype.

Final Design for iOS and Android

The final design being implemented in iOS and Android uses a drop down list that pops over the Edit Chapter display. This drop down popover list has a variable number of items depending on how many actions are valid in the particular circumstances. This also avoids the need to ask any additional questions of the user during the action (which is what the FileMaker prototype does). So the above table is now obsolete. The final table of allowable actions is held in a spreadsheet file separate from this design document (eventually it may be incorporated into this document); the spreadsheet file name is **`KIT Design Document Popovers.ods`**.

Implementation of Key It

As of September 2019 we are proceeding towards implementation of Key It. Thus we reviewed both the target devices for Key It and the essential features of the user interface so that we will implement those and not be sidetracked by graphics or behaviour that are merely artifacts of the software (FileMaker) that was used to make the user interface prototype.

Review of target machines for Key It

When we first started to think about KIT (Key It, or Type It as it was called then) the goal in focus was to provide an app which would be able to generate USFM marked up Scripture files for use by Adapt It Mobile (AIM). This implied that we would target the same hardware as AIM and use the same development software as for AIM. AIM is being developed using Cordova PhoneGap with HTML, CSS, and JavaScript, in order to be deployed on mobile devices running Android, IOS, and possibly Windows Phone or other systems.

Since then discussion has ranged over a wider set of goals and target devices.

1. Provide USFM marked up Scripture files for AIM (the initial goal).
2. Keyboard Scripture on smartphones or tablets independently of whether the USFM files would be used by AIM.
3. Provide USFM files for AID (Adapt It desktop)
4. Keyboard Scripture on desktops or laptops independently of whether the USFM files would be used by AID.

Goals 1 and 2 would be best served by KIT running on the mobile devices of interest - smartphones and tablets.

Goal 3 could be served by KIT running on mobile devices or, better still - for desktop or laptop users - on desktops or laptops (to avoid requiring a desktop or laptop user to purchase a mobile device).

Goal 4 would require implementing KIT on desktops and laptops.

The Cordova PhoneGap development system could be used for Goals 1 and 2. But Cordova does not generate apps for desktops or laptops, so if we were to extend our goals to 3 and 4 we would have to either find a different cross platform development system or else use two separate code bases - mobile devices vs desktops & laptops.

Thus an important step in proceeding with implementation is to choose our goals.

As of 28th Aug 2019 our primary intent is Goals 1 and 2. Goal 3 may become relevant later, but Goal 4 is probably not of interest because there are plenty of existing Scripture keyboarding apps that could be used.

As of 30th January, 2024 it is our intention to support the following:-

iOS from version 16 kitsui
iOS from version 11 kitios 1.0

Android from v 5.0 (API 21); but PlayStore requires targeting API 33 or later

Essential Data Items to be Stored

The data storage technique proposed for KIT is relational database. The central one to many relationships are

Bible >> Books
Books >> Chapters
Chapters >> Verses

Item Records

Each of the 31,102 verses in the Bible needs to be stored by KIT. When Scripture keyboarding starts, KIT needs to have all these verse records already created but empty of text. Each of these verse records must have a foreign key that links it to its chapter. A foreign key that links it to its book is not strictly necessary but may be helpful in some internal operations of KIT.

Most users of KIT would also have present the 116 extra Item records for Psalm ascriptions. Only if the user indicated when setting up KIT that the French Psalm verse numbering would used, would these 116 Item records be omitted.

In addition to the Item records storing the text of verses (and possibly Psalm ascriptions), other Item records are needed for the publication matters (headings, book titles, etc.). So every Item record needs an ItemType field indicating its type, because the details of how they are displayed and the actions allowed on them vary according to their type.

Chapter Records

Each of the 1,189 chapter records needs to have a foreign key that links it to its book. Each Chapter record stores its chapter number.

Book Records

Each of the 66 book records (more if the apochrypha were included) could have a foreign key linking it to its Bible record. Each Book record stores its book name.

Bible Record

For KIT as a standalone, single user app, there would be only one Bible record. In this case a one to many relationship between the only Bible record and all the Book records would not be necessary because all the Book records belong to the one Bible record; in this case a foreign key in each Book record would not be needed.

But if the abilities of KIT were extended to allow keyboarding more than one Scripture translation on the same computing device, there could be more than one Bible record and the Chapter records would need foreign keys linking them to their Bible record. In this case, the complete set of books, chapters and verses would probably be generated (from a template) when the user tells KIT to set up a new Bible translation.

In either case, the Bible record for a translation would store data pertaining to the whole translation. This will include the language code of the language of the translation, language related matters such as fonts and punctuation lists; possibly also user preferences for keyboarding that Scripture translation.

Essential Features of the Key It User Interface

The list of essential features below should be read in conjunction with *Thoughts Behind This Interface Design*.

The following descriptions are written from the point of view of how the user interface looks to the user. This is not necessarily the same as the internal details of how the user interface is finally implemented.

Vertical scrolling list of records

The central key feature of the prototype user interface is a vertically scrolling list of Item records that are editable. Editing focus is on only one record at a time.

Sorted in printed order

These Item records are displayed in the order they appear in printed Scripture. The reason for this order is that the user's primary task is to keyboard Scripture from a printed page and the Items must be displayed in this order.

Labelled by type for the user

The type of each Item record must be labeled for the user in terminology that the user understands. In some cases the labelling will match the internal type of the record, but in some cases two different types are, from the user's point of view, the same thing, and we should not bother the user with distinctions that matter only to the internal workings of KIT. For example, paragraph breaks are only ever paragraph breaks from the user's point of view, even though KIT needs to know whether the type of a paragraph break is a before-verse break or a within-verse break. But a verse and a verse continuation after an internal paragraph break need to be labelled differently to avoid confusing the user with two occurrences of the same verse number (observe the UIPrototype for a suggested wording).

Books, chapters and verses are fixed

Key It must provide a complete set of books, chapters and verses and must not allow users to add or remove books, chapters or verses (except as described in the next point).

Bridging and Unbridging of Verses must be provided for

Adjacent verses must sometimes, for reasons of grammar or discourse, be joined together. This alters the number of verse Items in a chapter but Key It must strictly control how this happens in order to maintain the integrity of the whole set of books, chapters and verses. Although a natural way to store Key It data is a relational database, Key It must not allow users access to the usual general purpose record creation and deletion facilities of relational databases.

Publication Items may be created or deleted as needed

Publication Items such as paragraph breaks, subject headings, book titles and book introductions, etc. may or may not be present in a Scripture publication. Key It must allow the creation and deletion of these Items fairly freely except for some essential constraints such as:

- book titles and introductions are allowed only before verse 1 of chapter 1 of each book
- parallel references should occur only after section headings

- paragraph breaks must not occur within bridged verses (if grammar and/or semantics has required the verses to be bridged, it makes no sense to split the bridge by a paragraph break)

Paragraph breaks in verses must be allowed

Occasionally a paragraph break needs to be within a verse, and Key It must ensure that the integrity of the chapters and verses is maintained. Key It should use VersCont Items for this but these should be shown to users in a manner that makes sense to users.

UI Controls

Because smartphones have small screens, the control buttons for users should be not be shown until the user taps on the description of the Item; tapping it should show the control buttons in an overlay that does not obscure too much of the display. When the user taps on one of these buttons the overlay should disappear and the action should happen.

Only relevant UI controls should be shown

When an overlay with control buttons is shown to the user, only buttons whose actions are allowable in that context should be shown. Furthermore the labelling of the buttons shown should be relevant to the context. These two ideas go a long way to preventing the user from doing things that are not allowable, and helping the user choose things that are allowable.

Additional questions for user

When KIT needs to ask a question of the user (such as whether a paragraph break should be before the verse or within the verse), the question and two buttons for the answer should be within the popover space and overwriting the buttons first shown in the popover space. This is to avoid taking up extra screen space for the question and answer.

Help user choose the Book and Chapter

When the user launches KIT, means should be provided so the user can choose which book and which chapter to edit.

An improvement to this is to program KIT to resume on the same book, chapter and verse as was being edited when KIT was last shut down. Most apps for mobile devices behave like this and it is what most users would expect. But while editing is in progress the user needs to be able to change to any other book, chapter, or verse.

Help user export a chapter to USFM

The user must be able to easily select a chapter of a book and export that chapter to USFM. The user must be able to easily select a book and export the whole book to USFM.

KIT Database Specification

The definitions of data tables and fields below focuses on the fields essential to the database structure of KIT. The UIPrototype has some additional fields (some of which are FileMaker calculation fields) that are needed for operation of KIT in the FileMaker context; implementation on Android and iOS will still need these facilities but will probably implement them by other methods.

DRAFT changes from integer IDs for records to a system of text IDs that can be created on collaborating devices running KIT that are without Wifi or Internet connection for some of the time.

Table: Bibles

Fields:

- BibleID - Int - Primary Key // ID number that Books, etc. link to
- BibleID – Text – Primary Key // 3 character code assigned to this Bible for this project
- Name - Text // Name to identify the Bible translation
- BookRecsCreated - Boolean // true if the Books records have been created
- CurrBook - Int // BookID of the currently selected Book

Note: KIT v1 will have only one Bible record as KIT v1 will be limited to one translation at a time. When we move on to v2 we may allow more than one translation in KIT at a time and then there would be one Bible record for each translation.

Table: Books

Fields:

- BookID - Int - Primary Key // the UBS standard Bible Book number
- BibleID - Int - Foreign Key // to link to the Bible record
- BookCode - Text // the UBS standard 3 character book abbreviation
- BookName - Text // the name of the book in the language of the Bible translation
- ChapRecsCreated - Boolean // true if the Chapter records for this Book have been created
- NumChaps - Int // number of chapters in the book
- CurrChapter - Int // ChapterID of the currently selected Chapter

Table: Chapters

Fields:

- ChapterID - Int - Primary Key // Serial number in order of creation
- ChapterID – Text – Primary Key // BibleID+BookID+3digitChapterNumber
  (e.g. TokP19121 for TokPisin Bible, Psalm 121
- BibleID - Int - Foreign Key // to link to Bible record
- BookID - Int - Foreign Key // to link to Book record
- ChapterNumber - Int // Chapter number seen by users
- ItemRecsCreated - Boolean // true if the VerseItem records for this Chapter have been created
- NumVerses - Int // Number of Scripture verses in this Chapter (never gets altered, not even by bridging or unbridging)
- NumItems - Int // Number of VerseItems in this Chapter (includes publication items and so varies as a result of user actions)
- CurrItem - Int // ItemID of the currently selected VerseItem

Note that the ChapterID as a serial number in order of creation will work nicely with an SQLite database since SQLite is inherently a single user database kept on the local device. If at some time in the future Key It were to be re-purposed as a multi-user networked app it would be necessary to rework Key It with a different database engine and a different method of handling ChapterID (probably a GUID). A similar comment applies to VerseItemID.

Table: VerseItems

Fields:

- ItemID - Int - Primary Key // serial number in order of creation
- ItemID – Text – Primary Key // BibleID+BookID+3digitChapterNumber+3digitVerseNumber
  (e.g. TokP19121008 for TokPisin Bible, Psalm 121, verse 8
  TODO: This does not handle publication items! Maybe need GUIDs for this??

- ChapterID - Int - Foreign Key // link to the Chapter the VerseItem belongs to
- VerseNumber - Int // the verse number as seen by readers
- ItemType - Text // Type of the VerseItem:- "Verse", "Title", "InTitle", "InSubj", "InPara", "Heading", "ParlRef", "Para", "ParaCont", "VerseCont", “PsAscr”
- ItemOrder - Int // order calculated from the VerseNumber and the ItemType
- ItemText - Text // If ItemType is "Verse" or "VerseCont" this is the Text of the verse in this Bible translation; some types use this for other text such as headings, introductory information, etc.
- IntSeq - Int // used only by Introductory items to record the sequence in which they occur
- isStartOfBridge - Boolean // true if this item is the start of a bridge of verses
- lastVsOfBridge – Int // if isStartOfBridge is true, then Verse number of the last verse added to the bridge

See comment above about ChapterID.

Table: BridgeItems

Fields:

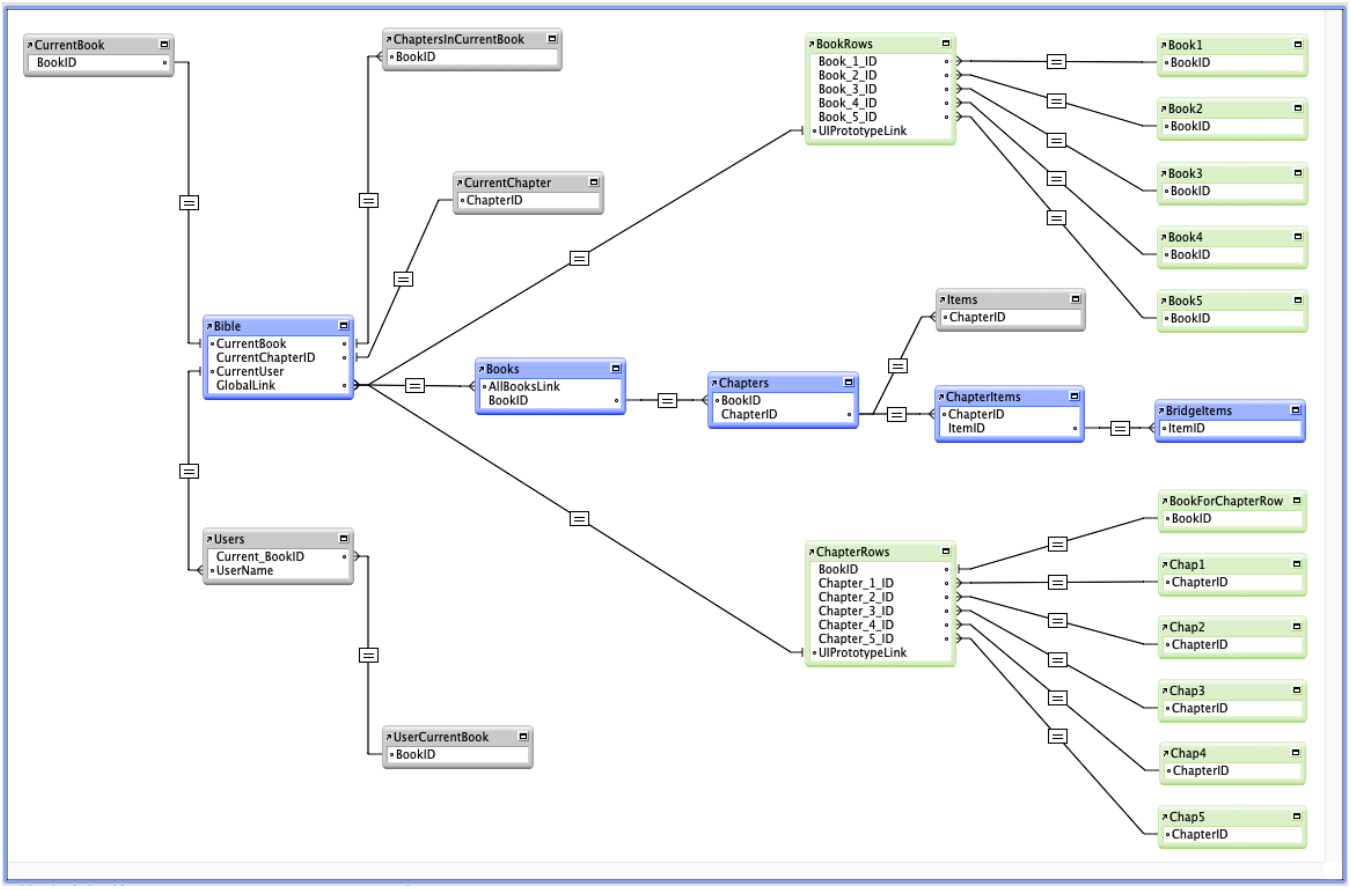
- BridgeID - Int - Primary Key // serial number in order of creation
- ItemID - Int - Foreign Key // link to the VerseItem this BridgeItem belongs to
- TextCurrentBridge - Text // Text of the current bridge
- TextExtraVerse - Text // Text of the verse added to the bridge; used when unwinding the bridge from right to left
- RecCreated // Timestamp of creation of this record (used to find the BridgeItem to be removed when un-bridging a bridge -- last one created for this bridge is the first to be deleted.)

For the FileMaker prototype it was very easy to use the record creation timestamps of BridgeItems records to determine which was the most recent BridgeItem for the current verse bridge. But that does not appear to be as easy when using SQLite. On the other hand using SQLite's automatically generated Primary Key (ROW_ID) will give the same result when the related BridgeItems records are sorted by descending ROW_ID – so KIT on iOS and Android do not need the RecCreated field.

Note that the BridgeID as a serial number in order of creation will work nicely with an SQLite database since SQLite is inherently a single user database kept on the local device. If at some time in the future Key It were to be re-purposed as a multi-user networked app it would be necessary to rework Key It with a different database engine and a different method of handling BridgeID (probably a GUID). In addition, it would be necessary to use creation timestamps of the BridgeItems records to identify which BridgeItem needs to be deleted during an unbridging operation.

FileMaker ERD

Here is the Entity-Relationships Diagram for the FileMaker prototype. The table occurrences coloured blue are the essential relationships for the data model. The table occurrences coloured light green are for a particular way of letting the user select a book or a chapter -- a method that would be done by relationships in FileMaker but which would be done quite differently in an Android or iOS app. The table occurrences coloured gray are convenience relationships for some operations on the data; in an Android or iOS app they may or may not be done by means of relationships.



Initial Data for Database

KIT_BooksSpec.txt and KIT_BooksNames.txt

One of the resources of the KIT app will be a text file, KIT_BooksSpec.txt, which lists all the books in the Bible, their 3 character abbreviations, and the numbers of verses in each chapter in the book.

Another resource of the KIT app will be a text file, KIT_BookNames.txt, which lists the 3 character abbreviations for the book together with their names in the language of the current localisation of KIT.

From these two text files (which amount to only 6KB) The entire Books -> Chapters -> VerseItems database can be created. This creation will be one of the steps in initialising the KIT app for a user to start work.

Each line of these text files would specify the data for a single Book; in some cases it would be a long line (e.g. the book of Psalms has 150 chapters). A short example is the book of Esther:-

KIT_BooksSpec.txt

17, EST, 22, 23, 15, 17, 14, 14, 10, 17, 32, 3

KIT_BooksNames.txt

17, Esther

These two single lines of text specify that Book Number 17, has the abbreviated name EST, and 10 chapters; the first chapter has 22 verses, the second chapter 23 verses, and so on. Furthermore that the name of the book in the English localisation of KIT is Esther.

When localised for use in Africa, KIT may well have

KIT_BooksNames.txt

17, Esta

so that a Swahili speaking user of KIT would see the book name with which he/she was familiar.

Notes

1) The Book Numbers are defined by the Bible Societies and all data files conforming to Bible Society specifications use the same identifying numbers for the books. This facilitates finding a Bible book in a data file or database because Book 17 is always the book of Esther regardless of how the name is spelled in the language of the translation being operated on.
2) KIT_BooksSpec.txt is allowed to have lines starting with the hash symbol and these lines are ignored during creation of the Books -> Chapters -> Verses database records. In the attached KIT_BooksSpec.txt the groupings of books are noted by comment lines.
3) KIT_BooksSpec.txt contains data that is independent of localisation of the app. KIT_BooksNames.txt has names of Biblical books that may be localised for different languages of users of KIT.

Psalm Ascriptions

The (long) line of text for the Book of Psalms shows which psalms have ascriptions. If the number of verses in a psalm is preceded by the letter 'A' then that psalm may have an ascription. Bible publications of the Psalms vary in how they handle Psalm ascriptions.

a) Some follow English Bibles by putting the ascription prior to verse 1 and not giving it a number;
b) Some follow French Bibles by incorporating the ascription into verse 1;
c) Some inconsistently behave like English Bibles for some Psalms but French Bibles for other Psalms.

So requiring the user to decide, at the very start of working with KIT, which approach is used in the printed translation being keyboarded from, is an unrealistic expectation on the user, especially as many KIT users will not have heard the term “Psalm ascription” and so KIT’s user interface would need to provide a lot of onscreen help (and there is very little space for onscreen help on a smartphone).

So it was decided to generate the VerseItem records for Psalms by providing a separate VerseItem for the ascription (if a given Psalm can have one) – so that the Book of Psalms will start out in KIT with every Psalm ascription known to Bible publishers. The “Ascription” descriptive label will (like the descriptive label on every displayed VerseItem) function as a button to display a popover with controls allowed on that item – in this case controls to Delete or Keep the ascription item. Thus the user of KIT can tailor the Psalm ascriptions to match the printed publication being keyboarded from.

Apocrypha

The books of the Apocrypha are not a focus of KIT and we anticipate most KIT users will not be interested in them. However, if a Christian group wants to include the apocrypha then there are two more files that could be included in the initialisation of KIT for use. These are KIT_ApocryphaSpec.txt and KIT_ApocryphaNames.txt. Neither of these files has been completed yet and we will probably only complete them if there is a serious request for them.

Lazy Creation and Initialisation of the Database

The above paragraphs are written on the assumption that the database for the entire Bibles -> Books -> Chapters -> Verses structure will be created and initialised before the user does any keyboarding.

But it would be possible to create and initialise the database records for one chapter at a time, and to do this on the first occasion that the user selects that chapter for editing. But the extent of "laziness" needs some care in design because the intention of the user interface design is that the user will always be selecting from a list of valid possibilities.

i. For KIT version 1, the single record Bibles table will be provided in a small database created before building the app and incorporated into the app's resources; on first launch of the app this small database would be copied into the app's documents directory and the 66 Books records would be created from the two text files in the app’s resources, otherwise the user would not be able to select a Book to edit.

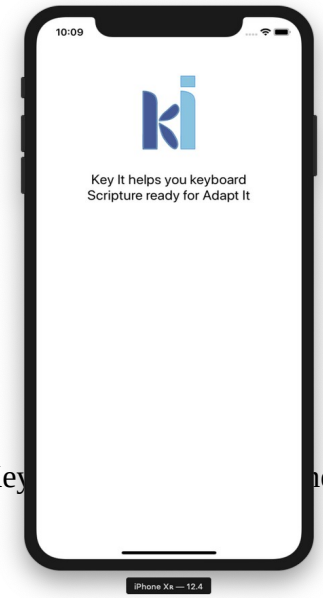
ii. On the first occasion that a book is selected for editing, the database records for the chapters of that book will need to be created, otherwise the user would never be able to select which chapter to edit.

iii. On the first occasion that a chapter is selected for editing, the database records for the verses in that chapter will need to be created, so that they can be presented as a vertically scrolling list ready for the user to keyboard their contents.

In the absence of lazy creation along these lines, the creation of 1,189 database records for chapters and 31,102 database records for verses would have to be done in one lengthy operation before the user could keyboard anything at all. But lazy creation along these lines will allow much smaller creation operations at the start of keyboarding work on a book, or a chapter. It is just the 66 database records for books that will need to be created in the very first launch of KIT.

Startup Sequence for KIT

There are numerous programmed actions involved in launching apps. Many of these are not noticed or understood by users of apps, but programmers must do some work to make or allow them to happen.



Key It Initial Launch on iOS

1. The KIT app icon must be visible and tappable. On iOS 11 this should be an icon 120 by 120 pixels.

2. A launch screen should be shown to the user. This should be a simple screen with not too many words and it should give the user reassurance that KIT is about to become available for use. Any initialisation that is needed should be done while the launch screen is showing. Apps that display a screen that looks like the app's working screen but is empty of data and in which nothing can be done while initialisation happens, can lead to user anxiety that the app is crashing.

3. On the first launch of KIT, copy the database containing the single Bibles record from resources to the app's Documents directory.

4. The first work screen to be shown to the user (after the launch screen) should allow the user to edit the name of the Bible. Other setup matters may later be included on this screen.

5. During launch of the app the bookRecsCreated flag is tested; if it is zero the 66 Books records are created so that the next work screen can show the list of Books. This normally happens only once (during the first launch of the app).

6. The second work screen to be shown to the user (after the launch screen) should have a scrolling list of the 66 books so that the user can tap on the name of the book to be keyboarded. This book then becomes the current book.

7. On the first time that this book is selected by the user, the database records for the chapters in that book need to be created.

8. The chapters in the current book should be shown in an array probably 5 columns wide with the device held in portrait orientation (but could be 10 columns wide if held in landscape orientation). For long books such as Psalms, Isaiah, etc. the array would need to be a scrolling array. Small icons or symbols can be used to indicate which chapters have already been keyboarded (helpful feedback to the user makes a big difference to user satisfaction with the app -- one reason why this graphical method of chapter choice is preferred). NOTE: I have looked at a number of Bible apps on smartphones and the only one that provided convenient choice of chapter or verse was one that used this graphical method. All the others had irritating disadvantages.

9. When the user taps on a chapter number, that chapter becomes the current chapter.

10. On the first time that this chapter is selected by the user, the database records for the VerseItems in that chapter must be created.

11. The VerseItems in the current chapter are then shown in a scrolling list of records. Each item in this scrolling list has

(i) a text field into which the user can type the text for that verse, and

(ii) a text label that displays the type of the VerseItem.
(In the FileMaker prototype this is a text button, at the top left of the text field, that displays the type of the verse item.)

12. When the user taps in the text edit space of a VerseItem, edit mode for that VerseItem is entered.

13. When the user does a tap-and-hold on the text of the VerseItem, an overlay appears above the VerseItem and this overlay has up to three buttons labelled for the only actions that are allowed on this particular VerseItem. An analysis of these allowable actions can be found at Allowable Actions on VerseItems.

Key It Initial Launch on Android

The startup sequence on Android will be essentially the same as on iOS. This section will be expanded as necessary to describe any differences needed for startup on Android.

Key It Startups After the First Launch

Because the database holds data for

- currBook
- currChapter of the current Book
- currItem of the current Chapter

subsequent launches of the app can go straight to the Book, Chapter, and Item that the user was last editing. Most of the time this will be what users want and it will facilitate keyboarding of the Bible. Note also that by recording the VerseItem (rather than a verse number) the user can be returned to editing a publication item (such as a subject heading) if that was what was being edited on last use of the app.

The navigation bar of the screen will have buttons to allow the user to select a different Book, or a different Chapter of the same book. There will also be a button to enable export of the USFM text of the chapter.

Because the database keeps a currChapter for each of the 66 Books (rather than a global currChapter data item), selecting a different Book that had been edited previously will automatically take the user to the Chapter of that book that was being edited the last time that Book was in focus.

Likewise, because the database keeps a currItem for each of the 1,189 Chapters (rather than a global currItem data item), selecting a different Chapter that had been edited previously will automatically take the user to the VerseItem of that Chapter that was being edited the last time that Chapter was in focus.

The currBook field in the Bibles record of the database is initialised to zero, so the very first time that a user starts keyboarding work, the app will take the user to the screen for selecting a Book.

The currChapter fields in Books records of the database are initialised to zero, so the very first time that a user starts work on a Book the app will take the user to the screen for selecting a Chapter of that Book.

The most common situation on launch of the app is that there will be a valid currBook and a valid currChapter. In this situation the user will be taken straight to the screen which displays the VerseItems in the current chapter of the current book.

There will usually be a valid currItem field for the current chapter and so the user will be taken to that VerseItem. If currItem is zero (or otherwise invalid) the user will be taken to the first VerseItem in that chapter.

User Interface Design of the App

The iOS implementation has been through several versions as system facilities were explored.

KIT03

At first the user interface was thought of as following the description of initial startup. On iOS this led to the choice of the iOS TabBarController as the top element. But this decision resulted in difficulty programming the changes from list of Books to list of Chapters to list of VerseItems, etc. The TabBarController is designed for an app with two or more different "modes" that the user switches between; thus it was difficult to program the changes.

KIT04

At that stage it was also realised that the part of the user interface to be most often shown to the user was the list of VerseItems in editing mode. That is the "scene" that would normally be the first shown to the user upon resuming keyboarding work. The "scenes" for choosing a Book and choosing a Chapter would only occasionally be shown and, after the user had dealt with them, they would disappear and the verse editing "scene" would be shown again. Similar thoughts apply to exporting a keyboarded chapter as USFM text:- switch to the Export "scene", deal with those details, then switch back to the verse editing "scene". (The word "scene" is used in the Xcode Interface Builder with the meaning implied here. The ideas in this paragraph also apply to an Android user interface for this app - except that Android documentation uses "activity" instead of "scene".)

The Navigation Controller class in iOS provides the abilities described above, including easy access to programming segues between scenes. So KIT04 will use a Navigation Controller (KIT03 uses a TabBar Controller).

The Navigation Controller provides (from iOS13 onwards) visual changes to subsequent scenes in a manner that is a little reminiscent of modal dialogs in menu + dialog interfaces; it provides straightforward ways of programming these segues, and also very easy unwinding back to earlier scenes.

KIT05

The design KIT04 worked to the extent of selecting Book, Chapter, and then going to the Edit Chapter scene, but it had some disadvantages.

1) The Edit Chapter scene needed code to detect whether it was being used during launch (Choose Book, then Choose Chapter), or when editing the VerseItems in the Chapter (its original intended use). This was achievable by putting two global flags on the AppDelegate. Global flags should never be used unless they are really, really necessary!
2) The segues through the scenes now had a "doubling back" in the diagram. Although very complex segue sequences are indeed possible, this "doubling back" should not really be necessary in a fairly simple app like KIT -- its presence suggests a poor design!
3) Adding another scene that launching passes through can solve both these problems. It would also provide a scene where the user can edit the name of the Bible and do other setup matters like choosing the writing system, etc. It could also have a flag to allow the user to choose to go straight to the current Book, Chapter, VerseItem read from the kdb.sqlite database once the user no longer wants to make changes to setup matters - launch code would still go through this scene but it would not be displayed to the user.

Software Design of the App

The SQLite database in the Documents directory will, as the user works on keyboarding, progress towards being a complete Bible -> Books -> Chapters -> VerseItems → BridgeItems data representation of the Bible being keyboarded.

At any time along the way, the SQLite database will have the data records for the VerseItems of the Chapters that have been keyboarded; it will have those Chapter records and the Book records that they belong to. As a result, whenever the app starts up it can read from the SQLite database any of those Books, Chapters, VerseItems and BridgeItems that have already been keyboarded.

While the app is running, its software will have a partial matching object instance hierarchy of
Bible -> current Book -> current Chapter -> current VerseItem.

- It will have all 66 Book records (so that the user can choose any one of those books to work on).
- It will have all the Chapter records of only the current Book that the user is working on (so that the user can choose any Chapter of the current Book to work on).
- It will have all the VerseItem records of the current Chapter of the current Book (so that the user can choose any VerseItem to work on).
- It will also have the BridgeItem records (if any) of the current Chapter of the current Book.

Although these lists of Book records, Chapter records, and VerseItem records will be kept as lists of data structs for the user to choose from, there will be single software instances of each of the following:

- Bible
- current Book
- current Chapter
- current VerseItem

and these instances will have member functions that handle interaction between the software instances and the user interface controllers.

Only one of the VerseItems of the current Chapter will be the current VerseItem. As the current VerseItem is edited, it is saved to both the list of VerseItems of the current Chapter and to the SQLite database before focus moves from it to another VerseItem; thus the user never needs to give a save command.

The user interface policy that the user will always be allowed to choose from valid lists of Books, Chapters, VerseItems means that there will be some "overlap" of software objects.

- The Bible instance must hold a list of valid Books that the user can choose from.
- The current Book instance must hold a list of valid Chapters of that Book to be shown to the user.
- The current Chapter instance must hold a list of valid VerseItems of that Chapter to be shown to the user.

This "overlap" increased the difficulty of thinking through the software design of this app! But it eliminates the need for error detection and user correction that would result from allowing the user to

- type a book name (with which a spelling error would produce a "no such book" error), or
- type a chapter number (a wrong number produces a "no such chapter error"), or
- type a verse number (a wrong number produces a "no such verse" error).

In addition to the lists of Books, Chapters, and VerseItems mentioned above, each of the iOS TableViewControllers that display these lists has its own internal list of table cells. But the TableViewControllers need to keep only the cells that fit in the currently displayed rows of the table (iOS also keeps a couple above and

below to improve smoothness of scrolling); thus if, on a particular screen, only 10 VerseItems fit vertically in the table, data about the remaining 160+ verses of Psalm 119 do not need to be kept by the TableViewController.

Furthermore, the data held by the iOS TableViewControllers needs only the data fields to be displayed to the user, rather than all of KIT's data fields about the items (internally, the TableViewControllers keep additional fields involved in iOS's display of the scrolling list).

Overview of Functions Involved in KIT iOS Launch Sequence

1. AppDelegate's didFinishLaunchingWithOptions() function

- On first launch creates an instance of the KITDAO class that encapsulates interactions between KIT and its SQLite database, and stores a reference to it for use by parts of KIT that need to operate on the database.
- Segues (automatically) to the Key It Setup scene (the storyboard's Entry Point).

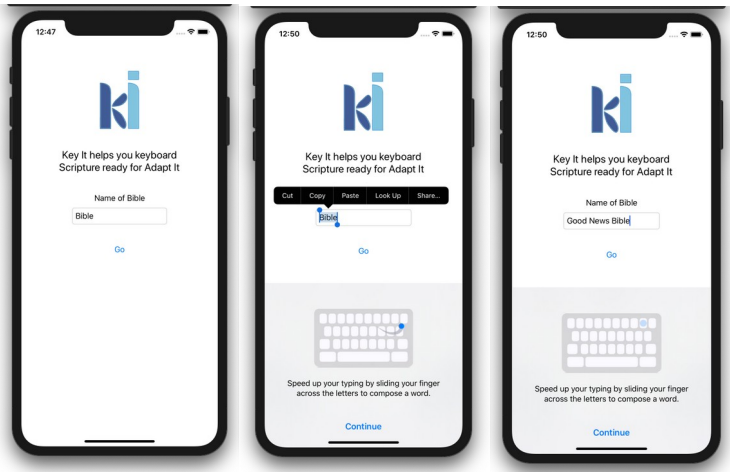
2. KeyItSetupController's viewDidLoad() function

- Reads the one and only Bible record from kdb.sqlite.

3. KeyItSetupController's viewDidAppear() function

- Initialises the Name of Bible text field in the Setup scene with the value read from the database.

4. KeyItSetupController's Go button (there must be a better name for this button!) function



- Removes the insertion point from the Name of Bible text field.
- Saves the (possibly edited) Bible name to kdb.sqlite.
- Creates an instance of the Bible class with the string from Name of Bible text field by calling the initialiser for the Bible class.

```
sqlite> SELECT * FROM Bibles;
1|Good News Bible|0|0
sqlite>
```

- Saves in the AppDelegate a reference to the Bible instance (other parts of the app need it).

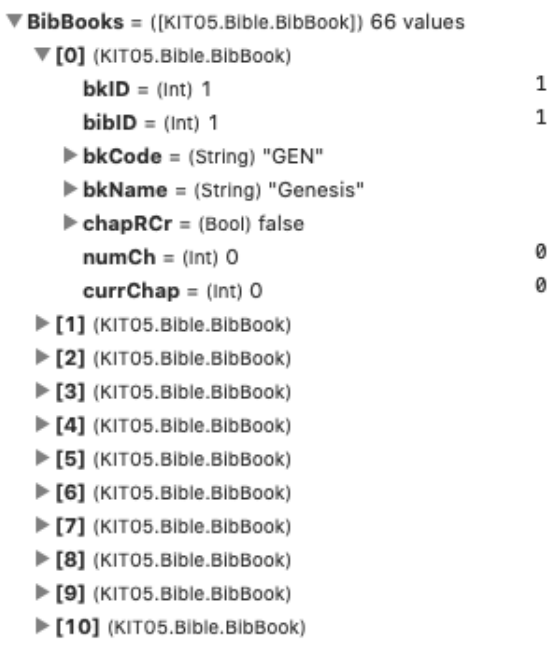
5. Bible class' init() function

- On the first launch of the app the Books records will not have been created, so it creates the 66 Books records based on the two resource text files KIT_BooksSpec.txt and KIT_BooksNames.txt; and then updates the Bible record in the database by setting the bookRecordsCreated (bkRCr) flag to true.

```
sqlite> SELECT * FROM Bibles;
1|Good News Bible|1|0
sqlite>
```

```
sqlite> SELECT * FROM Books;
1|1|GEN|Genesis|0|0|0
2|1|EXO|Exodus|0|0|0
3|1|LEV|Leviticus|0|0|0
4|1|NUM|Numbers|0|0|0
5|1|DEU|Deuteronomy|0|0|0
6|1|JOS|Joshua|0|0|0
7|1|JDG|Judges|0|0|0
8|1|RUT|Ruth|0|0|0
9|1|1SA|1 Samuel|0|0|0
10|1|2SA|2 Samuel|0|0|0
11|1|1KI|1 Kings|0|0|0
12|1|2KI|2 Kings|0|0|0
13|1|1CH|1 Chronicles|0|0|0
14|1|2CH|2 Chronicles|0|0|0
15|1|EZR|Ezra|0|0|0
16|1|NEH|Nehemiah|0|0|0
17|1|EST|Esther|0|0|0
18|1|JOB|Job|0|0|0
19|1|PSA|Psalms|0|0|0
20|1|PRO|Proverbs|0|0|0
21|1|ECC|Ecclesiastes|0|0|0
22|1|SNG|Song of Solomon|0|0|0
23|1|ISA|Isaiah|0|0|0
24|1|JER|Jeremiah|0|0|0
25|1|LAM|Lamentations|0|0|0
26|1|EZK|Ezekiel|0|0|0
27|1|DAN|Daniel|0|0|0
28|1|HOS|Hosea|0|0|0
```

- On every launch of the app, it reads the 66 Books records and builds the BibBooks array of structs that will be used in a number of ways as the app runs – especially to allow the user to choose which book to work on.



6. KeyItSetupController's Go button function (continued)

- Segues to the Select Book scene.

7. BooksTableViewController's viewDidLoad() function

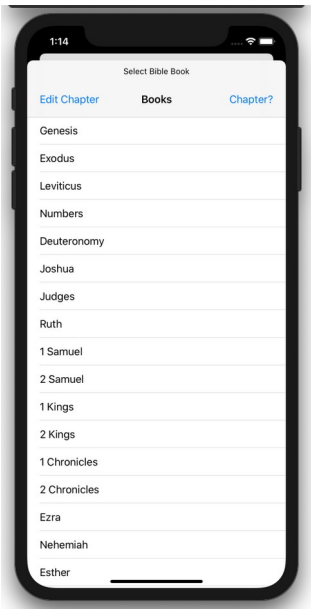
- Gets (from the AppDelegate) a reference to the Bible instance.

8. BooksTableViewController's tableView() functions (numberOfSections(), tableView(numberOfRowsInSection), tableView(cellForRowAt))

- These build the table view listing the 66 Books using the array that is part of the Bible instance.

9. BooksTableViewController's viewDidAppear() function

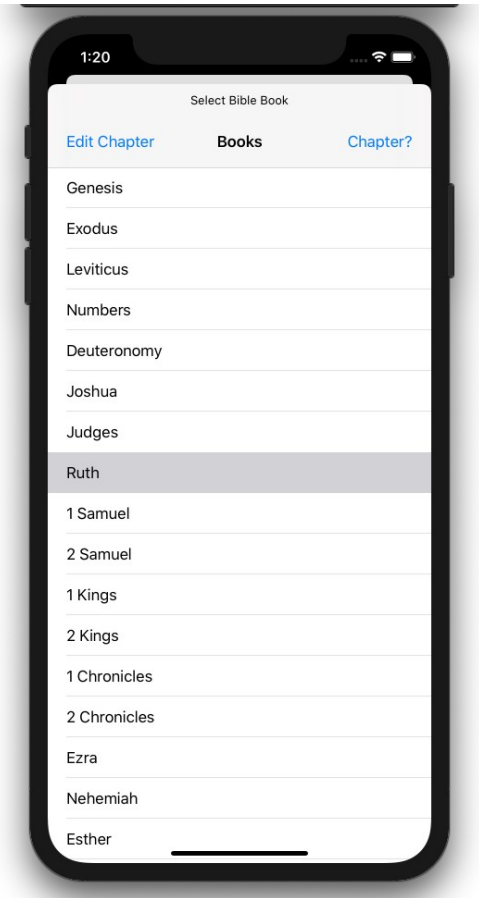
- If there is a current Book (as read from the Bible record in kdb.sqlite) it calls the Bible instance's goCurrentBook() function to select that as the current Book, create an instance of Book with the selected Book's details, and then segues to the Select Chapter scene.
- Otherwise waits for the user to select a Book from the list.



10. Book class' init() function

- On the first time a Book is made the current Book, its Chapter records will not have been created, so it creates the Chapter records based on the resource text file KIT_BooksSpec.txt; and then updates the Book record in the database by setting the chapterRecordsCreated (chapRCr) flag to true.
- On every occasion a Book is made the current Book, it reads the Chapter records for the Book and builds the array of structs that will be used in a number of ways as the app runs – especially to allow the user to choose which Chapter to work on.

11. BooksTableViewController's tableView(didSelectRowAt) function



- Gets from the array of Books the struct for the selected Book.
- Calls the Bible instance's setupCurrentBook() to make that the current Book, create an instance of Book with the selected Book's details, and update the Bible record of the database.
  In this Bible record the 1 after the Bible's name means that Book records have been created; and the 8 means that the current Book is book 8 (Ruth).
  In the Book record for Ruth the 5th field is true that the Chapter records have been created and that there are 4 chapters.
  In the Chapter records the number of verses are remembered and also the number of VerseItems (initially equal to the number of verses because no publication items have as yet been created).

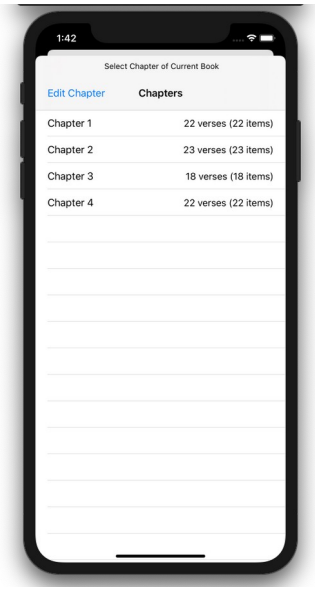
```
sqlite> SELECT * FROM Bibles;
1|Good News Bible|1|8
sqlite>
```

```
sqlite> SELECT * FROM Books;
1|1|GEN|Genesis|0|0|0
2|1|EXO|Exodus|0|0|0
3|1|LEV|Leviticus|0|0|0
4|1|NUM|Numbers|0|0|0
5|1|DEU|Deuteronomy|0|0|0
6|1|JOS|Joshua|0|0|0
7|1|JDG|Judges|0|0|0
8|1|RUT|Ruth|1|4|0
```

```
sqlite> SELECT * FROM Chapters;
1|1|8|1|0|22|22|0
2|1|8|2|0|23|23|0
3|1|8|3|0|18|18|0
4|1|8|4|0|22|22|0
sqlite>
```

- Segues to the Select Chapter scene.

12. ChaptersTableViewController's viewDidLoad() function



- Gets (from the AppDelegate) a reference to the Bible instance, and the current Book instance.

13. ChaptersTableViewController's tableView() functions

- These build the table view listing the Chapters of the current Book using the array that is part of the current Book instance.

14. ChaptersTableViewController's viewDidAppear() function

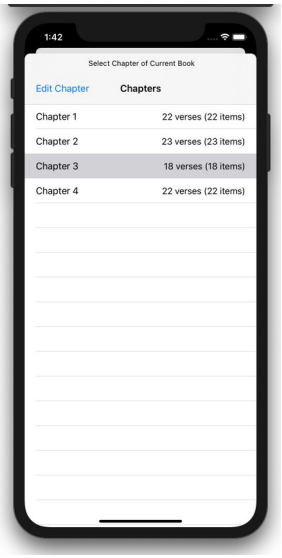
- If there is a current Chapter (as read from the current Book record in kdb.sqlite or set by BooksTableViewController) it calls the Book instance's goCurrentChapter() function to select that as the current Chapter, create an instance of Chapter with the selected Chapter's details, and then segues to the Edit Chapter scene.
- Otherwise waits for the user to select a Chapter from the list.

15. Chapter class' init() function

- On the first time a Chapter is made the current Chapter the VerseItem records for this Chapter will not have been created, so it creates the VerseItem records based on the number of Verses in the Chapter record; and then updates the Chapter record in the database by setting the verseItemRecordsCreated (itRCr) flag to true.
- On every time a Chapter is made the current Chapter it reads the VerseItem records for the current Chapter and builds the array of structs that will be used to display the VerseItems for the user to work on. Here are the VerseItem records for Ruth chapter 3.
  The Chapter record for Ruth chapter 3 has true for VerseItem records created.

```
sqlite> SELECT * FROM VerseItems;
1|3|1|Verse|100|Placeholder for v1|0|0
2|3|2|Verse|200|Placeholder for v2|0|0
3|3|3|Verse|300|Placeholder for v3|0|0
4|3|4|Verse|400|Placeholder for v4|0|0
5|3|5|Verse|500|Placeholder for v5|0|0
6|3|6|Verse|600|Placeholder for v6|0|0
7|3|7|Verse|700|Placeholder for v7|0|0
8|3|8|Verse|800|Placeholder for v8|0|0
9|3|9|Verse|900|Placeholder for v9|0|0
10|3|10|Verse|1000|Placeholder for v10|0|0
11|3|11|Verse|1100|Placeholder for v11|0|0
12|3|12|Verse|1200|Placeholder for v12|0|0
13|3|13|Verse|1300|Placeholder for v13|0|0
14|3|14|Verse|1400|Placeholder for v14|0|0
15|3|15|Verse|1500|Placeholder for v15|0|0
16|3|16|Verse|1600|Placeholder for v16|0|0
17|3|17|Verse|1700|Placeholder for v17|0|0
18|3|18|Verse|1800|Placeholder for v18|0|0
sqlite>
```

```
sqlite> SELECT * FROM Chapters;
1|1|8|1|0|22|22|0
2|1|8|2|0|23|23|0
3|1|8|3|1|18|18|0
4|1|8|4|0|22|22|0
sqlite>
```



15. ChaptersTableViewController's tableView(didSelectRowAt) function

- Gets from the array of Chapters of the current Book the struct for the selected Chapter.
- Calls the Book instance's setupCurrentChapter to make that the current Chapter, create an instance of Chapter with the selected Chapter's details, and updates the Book record to show the current Chapter. The Book record for Ruth is showing chapter 3 as the current Chapter.
- Segues to the Edit Chapter scene.

```
sqlite> SELECT * FROM Books;
1|1|GEN|Genesis|0|0|0
2|1|EXO|Exodus|0|0|0
3|1|LEV|Leviticus|0|0|0
4|1|NUM|Numbers|0|0|0
5|1|DEU|Deuteronomy|0|0|0
6|1|JOS|Joshua|0|0|0
7|1|JDG|Judges|0|0|0
8|1|RUT|Ruth|1|4|3
```

16. VersesTableViewController's viewDidLoad() function

- Gets (from the AppDelegate) a reference to the Bible instance, the current Book instance, and the current Chapter instance.

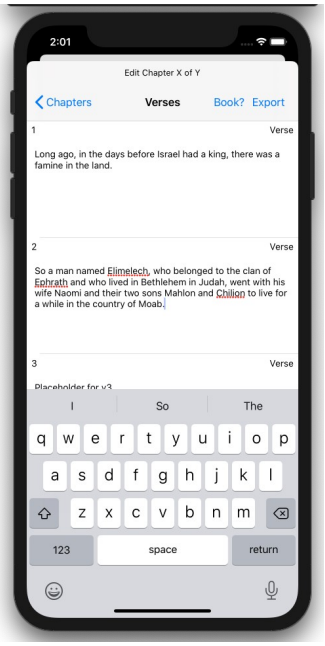
17. VersesTableViewController's tableView() functions

- These build the table view listing the VerseItems of the current Chapter of the current Book using the array that is part of the current Chapter instance.

18. VersesTableViewController's viewDidAppear() function

- Displays the VerseItems for the current Chapter of the current Book.
- If there is a current VerseItem for that Chapter, scrolls to make that VerseItem visible, and puts the insertion point into that VerseItem.

- Allows the user to edit the VerseItems of the selected Chapter of the selected Book.



Overview of Functions Involved in KIT Android Launch Sequence

1. SplashActivity's onCreate() function

- Create instance of KITDAO
- Save a reference to the instance in KITApp
- Save a reference to the app's resources in KITApp
- Post a method to be executed after 2sec to change to the SetupActivity

2. SetupActivity's onCreate() function

- Get the layout's Go button and Bible name edit text widget
- Set an OnClickListener for the Go button

3. SetupActivity's onResume() function

- Read the single Bible record from kdb.sqlite
- Set the BibleName into the EditText widget

4. SetupActivity's goButtonAction()

- Saves the (possibly edited) Bible name to the Bible record
- Creates an instance of the Bible class with the string from Name of Bible text field by calling the initialiser for the Bible class.
- Saves in KITApp a reference to the Bible instance (other parts of the app need it).

5. Bible class' init() function

- On the first launch of the app the Books records will not have been created, so it creates the 66 Books records based on the two resource text files KIT_BooksSpec.txt and KIT_BooksNames.txt; and then updates the Bible record in the database by setting the bookRecordsCreated (bkRCr) flag to true.
- On every launch of the app, it reads the 66 Books records and builds the BibBooks array of structs that will be used in a number of ways as the app runs – especially to allow the user to choose which book to work on.

6. SetupActivity's goButtonAction() (continued)

- Changes to the ChooseBookActivity.

7. ChooseBookActivity's onCreate()

- Gets the layout's widgets for Bible name and books list
- Sets an OnItemClickListener for when the user chooses a Book

8. ChooseBookActivity's onStart()

- If there is a current Book (as read from the Bible record in kdb.sqlite) and if the user is not choosing another Book, it calls the Bible instance's goCurrentBook() function to select that as the current Book, create an instance of Book with the selected Book's details, and then segues to the Select Chapter scene.
- Otherwise it sets up the ArrayAdapter with the list of Books and waits for the user to select a Book from the list.

9. ChooseBookActivity's chooseBookAction()

- Gets from the array of Books the struct for the selected Book.
- Calls the Bible instance's setupCurrentBook() to make that the current Book, create an instance of Book with the selected Book's details, and update the Bible record of the database.

- Changes to the ChooseChapterActivity

10. ChooseChapterActivity's onCreate()

-