vVote Developer Documentation

This document refers to the EAV (Electronically Assisted Voting) application that operates on tablet based devices. It also refers to the accompanying VPS (vVote Print Station) application that is built using the same method. It describes the software development and maintenance principles necessary to work on the application code.

# Introduction

The application is written as a single page web app, heavily utilising javascript to present “screens” to the user and to manage the underlying navigation, state and data model.

The application is logically designed as a multi-page, linear wizard that the user navigates through from left to right. It has the occasional popup screen to offer help and warnings where necessary.

The application has an alternative user interface that resembles an “Interactive Voice Response” (IVR) system much like you encounter over a telephone when calling a support hotline. The application plays audio instructions indicating your options and the available functionality. The options and functionality are triggered by touch interactions with the screen.

The visual representation of the application is given the name VUI – Visual User Interface. The IVR representation of the application is given the name AUI – Audio User Interface.

# Application Architecture & Design Philosophy

## Screen Navigation

**Screen 5**

**Screen 4**

**Screen 3**

**Screen 2**

**Screen 1**

3. “enter”

2. “exit”

1. “next screen”

**View Manager**

**Behavioural Aspect**

JavaScript

**Visual Aspect**

HTML

CSS

Eg.

legislative\_assembly\_candidate\_vote\_screen.html

legislative\_assembly\_view.js

legislative\_assembly\_visual\_screen.js

Every screen has a .html representation in the /views directory.

Every screen has an accompanying .js representation in /js/visual\_screens. It must inherit from the Screen class defined in screen.js.

ScreenFactory maintains a complete list of screen objects by name.

The accompanying .js is responsible for attaching and tearing down functionality and behaviour for its associated screen. It has an initialiser that gets called when it is entered, and an “exiter” that is called when it is exited.

Screen re-use is achieved through using the same buttons, help & warning screens, and attaching behaviour and content based on the context that it is used in. Help content is dynamically inserted into the help screens, and is often state dependent.

# How to add a new screen to the vVote application visual interface

1. Create a new screen in /views by copying one of the existing ones. Its content will be displayed within a holder DIV with a name matching the file name you used (step #3).
2. Name the file using the same standard naming convention as the other files in that directory. The name you give it is critical for successful addition.
3. Add it to the list of screens (DIVs) at the bottom of index.html using the given name.
4. Add it to the list of hidden screens in screens.css so that it will be hidden initially.
5. Either create triggers for it to be displayed if it is a popup, or stitch it into the application by having Next/Back buttons point to it from other screens.
6. Create a \_visual\_screen.js Screen class for it that defines entry and exit functions.
7. Potentially create a separate \_view.js class for it if it has lots of functional behaviour and the need for help/warning screens.
8. Add the new screen to ScreenFactory following the same convention as the other screens.
9. Add the .js files to index.html
10. Call VisualView.displaySection() on the new screen using the same convention as the other screens.

# How to add a new screen to the vVote application audio interface

Adding new screens to the AUI is a lot simpler, because there is only audio being played, there is very little dynamic behaviour.

1. Create a \_audio\_screen.js AudioScreen class for it that defines at least the initialEntry behaviour, but functions for any other desired behaviour as well. This includes alternative audio that plays when the screen is entered from other screens and requires non-introductory audio to be played.
2. Audio snippets are combined to construct complete paragraphs using the audio clip naming convention used in audio\_config.txt.
3. Add the new screen to AudioScreenFactory class following the same convention as the others.
4. Name it using the same standard naming convention. The precise name is critical for successful addition.
5. Insert the new screen into BlindInterfaceButtonNavigation and BlindInterfaceGestureNavigation using the same convention.
6. Define the actions that trigger the functionality within the screen, including navigating to other audio screens in the app.
7. Modify an existing screen’s actions to be able to navigate to the new screen. Ensure that you have covered all of the defined gesture/button options.
8. Ensure that your audio screen functions are called by AudioController at the appropriate time, and in the appropriate way.
9. It’s possible that the EO needs to switch from your new audio screen to a visual equivalent. You will need to find or create a matching visual screen, and write switchTo\_\_\_\_Screen function that defines how the screen should be constructed and switched to from the audio system to provide a user with a visual representation of the new audio screen.
10. Ensure that you don’t play audio and then leave silence for more than 5 seconds. You must continue playing audio in a repeated loop so that the user has actionable instructions at all times while using the application. The AudioInstructions class takes a callback to play the next audio once the current clip has stopped playing.

# Configuration

Various aspects of the application are configurable. Your changes to the application must maintain this configurability, so it’s essential that you understand which aspects of the app are configurable and how that configuration is achieved.

For the most part, configuration details are maintained in external files, usually in JSON format, and the manipulation is performed within the application using javascript.

## Internationalisation

The textual content and audible instructions in the visual system are required to be translated into foreign languages. To achieve this, the content is hot-swappable. To make the text hot-swappable, a library of HTML element IDs is maintained to convert the content of each element from one language to another.

This is not a completely satisfactory solution since some content is dynamically generated according to the state of the application. For instance, when warning users that they have not voted for the requisite number of candidates, a text string will be displayed containing the remaining number of candidates that they need to select. This manipulation is done dynamically at run time, and must be performed again whenever the language is changed.

So the library of international text strings often contains placeholders for the dynamic content.

Whenever you add new textual or audible instructions to the VUI, you must follow the convention for being able to provide alternative translations for each.

## Election Data

The election data used to construct the ballots within the application is provided in external files. These external files originate from the Nominations database and must be converted to JSON for use by the application. This functionality is outside the scope of the EAV application.

Within the application there are classes representing the various elements; candidates, parties, etc. These must be populated once the user has scanned their QR code. Their QR code dictates which electorate they belong to and what election data needs to be loaded for their ballots.

The election data also contains information on how ballots should be displayed, including whether ballot boxes are shown, candidates are ungrouped, races are undisputed, etc.

# Design Patterns

The application has been built with flexibility, re-usability and configurability in mind. To achieve this, separation of concerns has been applied, and re-applied as the app has evolved in different directions.

The following patterns have been used and their use is explained.

## Dependency Injection

Variable scope in javascript applications means that globally defined variables aren’t necessarily available within scope in particular blocks of code. Global variables are to be avoided as best as possible regardless.

In their place is utilisation of Dependency Injection – the ability to instantiate/re-instantiate an object from anywhere in the code. A function is called – getContainer() – to retrieve the injector (or container) that holds all of the instance variables. In this application, just one instance of each variable is maintained, and so .Reused() is utilised when registering objects in the D.I container. The object can then be used as needed, and it maintains its state throughout the lifecycle of the application.

## Factory Pattern

A Factory is another way of maintaining instance variables for use as and when needed. Instances are retrieved by their type. In this app, we maintain instances of screens by their type within factories, to be retrieved as and when they are needed.

## Command Pattern

A Command is a way of encapsulating a piece of functionality and attributing it to a trigger, so that the functionality is called when the trigger is provided. By separating the action from the trigger, it is possible to quickly and easily swap behaviours from one trigger to another. This is seen utilised in the audio UI where gestures and buttons in the touch interface are used to trigger a range of behaviours and interactivity in the application.

## Model/View/Controller

The application utilises the traditional MVC pattern to separate program control, presentation and data management. The views are either defined visually by HTML/CSS or audibly by MP3. This has been discussed above.

Both the AUI and VUI have controller classes that are used to manage the presentation layer. They read the data layer and determine what needs to be presented in the UI in each instance. The controllers update the underlying data model according to the user’s interactions, which includes things like moving between candidates, selecting them, etc.

The AUI and VUI also have manager classes AudioScreenManager and VisualScreenManager that manage the state of the screens in the presentation layer.

The data model is represented by a range of classes including BelowTheLineCouncilBallot, AboveTheLineCouncilBallot, LegislativeAssemblyBallot, AssemblyCandidate, CouncilCandidate.

## Delegate Pattern

There are also over-arching Manager classes that manage different aspects of the model. The BallotManager is responsible for delegating navigation and ballot interaction to the relevant ballot model classes. The OptionsManager is responsible for delegating navigation and options menu interaction to relevant parts of the application.

# Master Bulletin Board (MBB)

EAV communicates with a back end server known as the MBB. The MBB is responsible for the verification, authorisation and specialised administration of generating voting ballots, submitting votes, cancelling votes, etc. EAV treats the MBB as a remote service, and calls service endpoints via asynchronous AJAX calls. It waits for the response in most scenarios before continuing and updating the UI, else it reports an error in the UI. It’s important for the end user to know that the actions completed while communicating with the MBB are completed following ACID principles.

The data sent between EAV and the MBB is used to verify that correct vote submission is occurring; i.e. no repeat submissions, or submitting cancelled candidate slips or invalid slips, etc.

The MBB ultimately stores all of the votes, and updates the public interface Web Bulletin Board (WBB) with this information on a regular (nightly) basis.

## Printing

The applications (both VPS and EAV) print out an “infinity length” document. These printouts represent the Candidate List, and a matching Preferences Receipt. The layout and positioning of these two slips is imperative. They must match up when they are lined up side by side. This is for the voter’s visual verification that their vote has been submitted correctly.

Very few technologies offered the ability to “infinity print” these slips with tight controls over positioning, sizing, etc. The chosen solution is a javascript library that generates a PDF document dynamically. It does this by generating rows of content and appending it to a manufactured document. All of this is contained within the source file ballot\_generation.js.

## Shuffle Order

There are three different orderings of the candidates;

1. Ballot order (or legal order). The candidates are given a specific ordering that is presented to us in the Nominations data.
2. Preference order. The order (first to last) that the voter has selected.
3. Shuffle order. The order that the candidates are presented on the printed slips.

The application often must transform the candidate lists from one ordering to another. It is imperative to maintain the correct shuffle order of the candidates, which is controlled by two things – the ballot order of the candidates and the shuffle code randomly generated by the MBB.

Of major importance in the code is whether zero-based indexing arrays are being used, or one-based indexing. Ballot position is determined by data provided in the Nominations data. The preference order numbering is determined in the core model classes; BelowTheLineCouncilBallot, AboveTheLineCouncilBallot, LegislativeAssemblyBallot. The shuffle ordering is determined by functions in the core model classes, and by the shuffle code provided by the MBB.

If any one of these factors changes, then the shuffle order will be broken, and the printed output will be incorrect.

Zero-based indexing issues could be resolved by finding or creating a suitable Map ADT that handles shuffle order transformations. The ballotPosition generally is one-based, and the shuffle order is generally zero-based, causing the primary issue. Either of these could change at a moment’s notice, but probably won’t.

## Switching between AUI and VUI

Another challenging aspect of the application is the ability to switch immediately from the visual representation to the audio representation and vice versa without any loss of state. What makes this challenging is the differences between the VUI and AUI. The screen representations may not perfectly match up, or the way things are represented are not identical. But it also means that state must be carefully maintained, monitored and reset throughout the application lifecycle.

There are two main paths for switching from the AUI to the VUI; in the options menu, and by pressing the two visible buttons on the screen at the same time. These two methods have different meanings. The two buttons are for a sighted person to assist a blind person by seeing where they are, what they have done, and explaining what they need to do, before switching back to the audio system. The options menu in the AUI lets the voter quit the AUI and start with the VUI. It maintains their votes.

There are two main paths for switching from the VUI to the AUI; at the start of the VUI on the settings screen there is a button to allow the voter to switch to the AUI. This means they have given up trying to use the VUI, and their votes are maintained. The other method is in the help screen of the VUI; a button is presented only when the EO has switched from AUI to VUI by pressing the two buttons in the AUI. It takes the voter back to the AUI screen they were in before seeking assistance, so that they may continue their voting session.

By abstracting away the representation of the core model, we are able to reuse the code to manage candidate selection, party membership, ballot position, etc. That means we are more easily able to switch from the VUI to the AUI without ill effect and maintain consistency.

## Adding a new language to the app

Languages can be added to the application through configuration files. The two files you need to modify are audio\_config.txt and language\_config.txt. The audio config file is needed locate the audio files representing the translated audio instructions for the VIS interface.

The language config file needs to have a new entry under “languages” that follows the same convention as the others. It needs to nominate the dictionary in the same file that will be used to translate text strings in the app. The language dictionary will need to be added, and will need to have a mapping for every ID tag listed in “conversion\_labels”.