User-Designed Background Tasks in App Inventor

by

Vedha Sayyaparaju

S.B., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

©Massachusetts Institute of Technology 2015. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

January 30th, 2015

Certified by

Harold Abelson

Class of 1922 Professor of Computer Science and Engineering

Thesis Supervisor

Accepted by

Albert R. Meyer

Chairman, Masters of Engineering Thesis Committee
Abstract

In this thesis, I describe how I designed and built multiple components and extensions to App Inventor 2 that will allow application builders to create custom services and background tasks and to build applications that can interact with these services and tasks. Previously, the App Inventor platform only supported the creation of applications which had a screen in the foreground at all times. As such, the main abstraction of App Inventor was this notion of a “Screen”. These screens could launch certain tasks to run in the background, but they were limited to the few tasks that were exposed by the App Inventor interface. Application builders could not design and customize their own background tasks. This restricted App Inventor users from building certain types of applications, for example, a music player application or an application that has heavy network communication. To enable users to build such applications, I extended the App Inventor platform to expose a “Task” object in addition to the existing “Screen” object. I created a messaging system which would allow Screens and Tasks to communicate with each other. I also developed additional task components that could be contained in these new Task objects. Users can customize the functionality of Tasks by putting together multiple task components. In this way, App Inventor users can now build more functional applications and explore a part of the Android SDK that was previously out of reach.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering
Dedication

I would like to dedicate this to my mom and dad who have always given me their unconditional support.


Acknowledgements

I would like to thank my thesis advisor Hal Abelson, for supporting me throughout my Undergraduate Advanced Project and my graduate career. A huge thanks to Andrew McKinney, who has mentored me throughout my work on my graduate research and spent countless hours discussing and providing feedback on my work. I would also like to give a special mention to Jeffrey Schiller who helped me navigate the confusing world of App Inventor and guided me through my design process. Marisol Diaz, who is the heart and soul of App Inventor, deserves a huge shoutout. Thank you Marisol for supporting me throughout my time with App Inventor. Finally, thanks to my parents, Sridevi and Bhaskar Sayyaparaju, for the moral support you have provided throughout this process.
# Contents

1 Introduction ............................................. 9
   1.1 Purpose ............................................. 9
   1.2 Contributions ............................... 10
   1.3 Interface ......................................... 12
   1.4 Simple Example ....................... 15
   1.5 Outline ........................................... 16

2 Background ............................................. 17
   2.1 Terminology .......................................... 17
   2.2 General Architecture ................... 18
   2.3 Web Application ....................... 18
   2.4 Yail Generation ............................... 19
   2.5 Kawa Runtime and compilation .......... 19

3 Background Tasks ............................................. 20
   3.1 Motivation ........................................... 20
   3.2 Intents .............................................. 21
   3.3 Android Services ........................ 21
   3.4 Considerations for AI2 .................. 23
   3.5 Top-Level Components .................. 24
   3.6 Tasks Interface .............................. 24
   3.7 Task Implementation .................... 25
   3.8 Web Application ............................... 27
      3.8.1 File Editors .................................... 27
      3.8.2 Adding Tasks .............................. 28
      3.8.3 Miscellaneous Changes ..................... 29
   3.9 Form Changes ....................................... 29
   3.10 Blocks to App ................................. 30
      3.10.1 Generating YAIL .................... 31
      3.10.2 Tasks in Kawa ...................... 32
      3.10.3 Kawa Runtime Changes ............. 33
      3.10.4 Block Context ....................... 35
      3.10.5 Android Manifest .................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 ApplicationMessenger</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Broadcast Receivers</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Security</td>
<td>38</td>
</tr>
<tr>
<td>4.3 Specifications</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 Properties</td>
<td>39</td>
</tr>
<tr>
<td>4.3.2 Methods</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3 Events</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Implementation</td>
<td>40</td>
</tr>
<tr>
<td>4.4.1 Android Support Libraries</td>
<td>41</td>
</tr>
<tr>
<td>4.4.2 Defining Messages</td>
<td>41</td>
</tr>
<tr>
<td>4.4.3 Receiving Messages</td>
<td>42</td>
</tr>
<tr>
<td>4.4.4 Sending Messages</td>
<td>43</td>
</tr>
<tr>
<td>5 Notifications Component</td>
<td>44</td>
</tr>
<tr>
<td>5.1 Background</td>
<td>45</td>
</tr>
<tr>
<td>5.2 Specifications</td>
<td>46</td>
</tr>
<tr>
<td>5.2.1 Methods</td>
<td>46</td>
</tr>
<tr>
<td>5.3 Implementation</td>
<td>47</td>
</tr>
<tr>
<td>6 Base Task Component</td>
<td>49</td>
</tr>
<tr>
<td>6.1 Task Component</td>
<td>50</td>
</tr>
<tr>
<td>6.2 New Task Components</td>
<td>51</td>
</tr>
<tr>
<td>6.3 General Guidelines</td>
<td>53</td>
</tr>
<tr>
<td>7 Sample Applications</td>
<td>55</td>
</tr>
<tr>
<td>7.1 File Application</td>
<td>55</td>
</tr>
<tr>
<td>7.2 Nearby Application</td>
<td>57</td>
</tr>
<tr>
<td>8 Future Work</td>
<td>62</td>
</tr>
<tr>
<td>8.1 Live Testing</td>
<td>62</td>
</tr>
<tr>
<td>8.2 Error Handling</td>
<td>62</td>
</tr>
<tr>
<td>8.3 Blocks and Context</td>
<td>63</td>
</tr>
<tr>
<td>8.4 More Components</td>
<td>63</td>
</tr>
<tr>
<td>9 Conclusion</td>
<td>64</td>
</tr>
<tr>
<td>A Blocks to Yail</td>
<td>65</td>
</tr>
<tr>
<td>B Updated Task Components</td>
<td>68</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Screen1 designer view ................................................. 13
1.2 Screen1 blocks ......................................................... 13
1.3 Task1 designer view ..................................................... 14
1.4 Task1 blocks .......................................................... 14
1.5 Task1 blocks for quotes application ................................. 15
3.1 StartTask and StopTask methods ........................................ 25
3.2 TaskStarted event ......................................................... 25
3.3 Example of user-defined behavior for Task events ....................... 26
3.4 Task Designer view and its relation to the TaskEditor components .... 27
3.5 Example of starting and stopping a Task ................................. 31
3.6 Example YAIL code to add a component and define an event handler for that component .................................................. 31
4.1 ApplicationMessenger Properties ........................................ 39
4.2 ApplicationMessenger Methods .......................................... 40
4.3 ApplicationMessenger Events ............................................. 40
5.1 NotificationComponent methods ........................................ 46
7.1 Properties of the Web component ........................................ 56
7.2 Blocks for making the web request ........................................ 56
7.3 Properties of the ApplicationMessenger .................................. 57
7.4 Screen1 interacting with Task1 ............................................ 57
7.5 Button blocks for Screen1 ................................................ 58
7.6 Task1 Initialize blocks ..................................................... 58
7.7 Properties of the LocationSensor .......................................... 59
7.8 Event triggered on location update ........................................ 59
7.9 Procedure to call Places API ............................................ 60
7.10 Launching notification with response from API ......................... 60
7.11 Extracting information from response .................................... 61
A.1 Task1 designer view ...................................................... 65
A.2 Task1 blocks ........................................................... 65
A.3 Task1 json representation ................................................. 66
A.4 Task1 blocks in xml representation ....................................... 66
A.5 The yail for Task1 ........................................... 67
B.1 Changes in Web constructor ................................. 68
B.2 Error dispatch changes in Web component ............ 69
B.3 Handler changes in Web component ..................... 69
Chapter 1

Introduction

1.1 Purpose

MIT App Inventor is a blocks based programming language designed to enable people of all programming levels — even novices — to build fully functional Android applications. Previously, App Inventor only supported applications which had a screen in the foreground at all times. It was not possible to decouple tasks from the screen that they were associated with. This limited the types of applications users could build.

Music players, heavy communication with the network, and many other use cases do not fit well within this model. A music player, for example, should continue to play music, without disruption, even when a user switches between applications or between screens within an application. If an App Inventor user must associate the entity playing music with a particular screen, the user experience will be compromised. When a user wants to perform certain communication over the network, it would be useful to separate this communication from the screens. Then, the users interaction with the screen will not inhibit the network activity and the network communication will not slow down user interaction with the screen. There are many other use cases — some will be described in detail later in this paper — which are not well supported by the current App Inventor model.

The purpose of my thesis is to develop an ecosystem that allows App Inventor
users to define custom background tasks which are decoupled from screens. Once a user has this ability to define tasks which are separate from screens, it is important to open a channel of communication between these tasks and screens. After all, the user interacts directly with the screens and not the tasks. Therefore, in addition to allowing users to define their own tasks, this thesis will allow screens and tasks to communicate with each other. This thesis will also give an App Inventor user more control over notifications, so that tasks can communicate with app users through notifications. Since tasks don’t have their own screen or user interface, it is important to enable as simple a user experience as possible.

1.2 Contributions

To create the ecosystem described above, I have made the following contributions:

1. **Tasks**

   As mentioned previously, an App Inventor application is made up of multiple Screens which interact with each other. For each Screen, the App Inventor user designs the user interface and specifies the user interaction. I introduced Tasks to App Inventor, so now applications are made up of Screens and Tasks. These Tasks and Screens can interact with each other to make a fluid application. Unlike Screens, Tasks have no user interface. Tasks perform user specified activities in the background. For example, a user can design a Task which downloads and saves a file every 5 minutes. By defining this behavior in a Task instead of a Screen, the app developer ensures that this action is performed even when the application is not currently in use. A Tasks in App Inventor is an abstraction of the Android Service, just as a Screen in App Inventor is an abstraction of an Android Activity.

2. **ApplicationMessenger**
Screens have a mechanism to communicate with each other. One screen can launch another screen and send along any information that might be useful. With the new concept of Tasks, it is important to make sure all parts of the application can still talk to each other. Especially because tasks have no user interface themselves, it is important that any important information from the Tasks is somehow conveyed to the application user. As such, Tasks must be able to communicate with Screens, which have their own user interface. For this purpose, I have created the ApplicationMessenger component. This component is an abstraction to the Android LocalBroadcastManager. Basically, this component supports the publish-subscribe model in Android. Screens and Tasks can both subscribe to certain events and determine what actions to perform when those events occur. They can also broadcast events which will trigger any subscriber’s actions.

3. **Notifications component**

The Notifications component is not directly necessary to enable Tasks in App Inventor. However, using notifications makes error handling with Tasks much cleaner. For Screens, error messages can be easily shown by pulling up a pop-up dialog. However, for Tasks, with no user interface, the situation gets trickier. Tasks can run in the background even when their parent application is not currently in use. Therefore, there needs to be a mechanism to communicate with the user from the Task even when there is no foreground Screen. Notifications are useful in this context. The Notifications component can be used from both Screens and Tasks, and can create and delete notifications with custom messages.

4. **Base Task component**

App Inventor previously supported two types of components: Visible and Non-Visible components. Visible components are UI elements that take up physical
space on a Screen. For example, a Label and a Button are Visible components. Non-Visible components do not take up space on the parent Screen. For example, the Camera component which launches the Camera does not actually embed a camera on a Screen. Instead, a Camera view, separate from the user-designed Screen, comes to the foreground when the Camera is launched. So while the Camera component doesn’t take up physical space on the Screen which launches it, the Camera eventually uses the phone screen real estate when it is launched. However, some Non-Visible components never take up screen space — on their parent Screen or any future foreground screen space. With Tasks, this distinction becomes very important. Because Tasks always do their work in the background, they cannot use components which will take up screen space at any point in the future. Therefore, I created a base class for components that can be used in Tasks. These components never effect the UI of an application and can be used in both Screens and Tasks. This new class is the NonVisibleTaskComponent.

1.3 Interface

The interface of App Inventor does not drastically change with the addition of Tasks. We try to adhere to App Inventor principals as closely as possible. Let’s take a look at what the interface will look like.

We’ll look at a simple application that has one Screen and one Task. Screen1 is the default Screen that is created with any App Inventor application. We will add a Task, called Task1, to the application by clicking on the “Add Task” button.

Screen1 has a Button to start Task1 and a Label to display the status of Task1 – whether it has been started or not. It also has an ApplicationMessenger component which communicates with Task1 and learns when Task1 has been started.
Task1 has an ApplicationMessenger to communicate with Screen1 and a NotificationComponent to send notifications. When the Task is started, Task1 launches a notification that informs the user it has started. Task1 also sends a message out using the ApplicationMessenger. Screen1 sees this message and updates the text in its Label, to indicate that the Task has started.
This was a very simple example to understand how the interface has changed with Tasks. Tasks are very powerful and can perform much more complicated tasks. If I add a File component to Task1, we can download a File and notify Screen1 when the File is downloaded. If we add a Web component to Task1, users can interact with any REST API from the Task. We can have an application in which users specify files to upload in Screen1, and the Web component performs the actual upload in Task1. This way, Screen1 does not get stuck on long network operations and the intensive work happens in the background in Tasks.
1.4 Simple Example

In this section, we will take a look at a simple application that can be built with Tasks that can’t be built using the current App Inventor platform. We will build an application that sends the user a notification with an inspirational quote every hour.

This app will have one Screen and one Task. The Screen has one button that starts the Task. The Task has a Timer, a Web component, and a NotificationComponent. The Timer is enabled when the Task is started. Whenever the timer fires, we will use the Web component to get an inspirational quote from forismatic.com by making a request to the url\texttt{http://api.forismatic.com/api/1.0/?method=\texttt{getQuote}\&\texttt{format=json}\&\texttt{lang=en}}. When we have received the result, we will launch a notification with this quote. The application does not need to be in the foreground for the notification to be launched. The user can interact with other application and will still receive these hourly inspirational quotes.

![Figure 1.5: Task1 blocks for quotes application](image)

Figure 1.5: Task1 blocks for quotes application
1.5 Outline

In Chapter 2, I give some background for this project. I describe at a high-level the architecture of App Inventor and some of the key terminology. In Chapter 3, I discuss in detail the changes made to add support for background tasks to the App Inventor platform. Chapter 4 describes the ApplicationMessenger Component, and how Screens and Tasks can communicate with each other. In Chapter 5, I talk about how the Notifications Component was built and how it can be used to enhance the Tasks ecosystem. Chapter 6 looks at the new type of component required for Tasks. Chapter 7 describes sample applications that I have developed, which demonstrate some use cases that could not previously be handled in App Inventor. In Chapter 8, I explore some future work that can be done to improve the Tasks ecosystem. Chapter 9 concludes this thesis.
Chapter 2

Background

Tasks are top-level components in App Inventor. They are as much a core component of the application as Screens. Therefore, the changes made to add Tasks to the platform cut across many parts of the codebase. It is therefore very essential to develop a good understanding of the architecture of App Inventor to understand the scope of this project.

2.1 Terminology

1. Screens
   A Screen is one of the main parts of an App Inventor application. An application is basically a collection of Screens which are designed by the App Inventor user, the application maker. The app maker designs the user interface of each screen and the behavior of each screen. The app maker also specifies the interaction between screens. For example, clicking on a button on one Screen could cause another Screen to launch.

2. Components
   Each Screen in App Inventor is made up of Components. An app developer designs the UI of a Screen by dragging and dropping multiple Components onto a Screen. For example, a simple Screen could be made up of a Button
Component and a Label Component. The user can then specify behavior for Components on the Screen. For example, when the Button is clicked, the Label can display the text, “Hello”.

2.2 General Architecture

Users interact with the App Inventor web application and create Screens which make up an application. For each Screen there is a Designer view and a Blocks view. The Designer view is where components are added to a Screen and a user specifies the physical layout of the Screen. In the Blocks view, the user describes the interaction between components by describing behavior upon certain events. Events are a very important concept in App Inventor. The functionality of the components themselves is written in Java by the Component developers. There is a Kawa bridge that acts to compile the designer and blocks view specifications into a packaged Android application. There is also a live testing mode in App Inventor, but the details of that are beyond the scope of this project.

2.3 Web Application

Every time the add Screen button is clicked, a Screen object is created which separately keeps track of a FileEditor and a BlocksEditor for the Screen. The FileEditor corresponds to the Designer view and keeps track of components that make up an application and their properties. The BlocksEditor keeps track of the events and interactions between these components.
2.4 Yail Generation

YAIL is an internal app inventor language that is used to represent the Screen and its behavior. The Screen and its components are represented by a simple JSON file with “.scm” extension. This is generated from the FileEditor representing the Screen. The BlocksEditor however generates a YAIL file with the “.yail” extension. This file when compiled will hook into the kawa runtime.

2.5 Kawa Runtime and compilation

During the compile stage, the Kawa runtime runs the yail file and generates the code for an Android application. The Android Manifest file also gets generated during this stage. Each screen is declared in the manifest and any necessary permissions are requested. The application is then packaged into a “.apk” file.
Chapter 3

Background Tasks

The following chapters detail the implementation of the task ecosystem in App Inventor. In this chapter, we start by delving into the changes made to add Tasks as top-level components — just as important as Screens in current App Inventor.

3.1 Motivation

As of January 4th, 2015, Pandora is fifth on the list of top free apps in the Google Play Store [1]. Music players like Pandora depend on Android Services to allow users to play music in the background while still interacting with other applications. Imagine what a horrible user experience would result if users were locked into the Pandora app to listen to music. Building applications that allow this functionality requires the use of Android Services.

Because App Inventor is an abstraction of the Android platform, app developers can only use those parts of Android which App Inventor has exposed. Previously, Screens abstracted Android Activities and were considered the main components of an App Inventor Android Applications. However, apps like Pandora rely on Android Services which allow them to perform operations in the background, separate from any foreground Activity. Because the current App Inventor platform does not expose Android Services, app developers are quite restricted in the types of applications
3.2 INTENTS

3.2.1 BACKGROUND TASKS

they can build.

To fix this issue, I decided to create Tasks, which expose Services as top-level components in the App Inventor platform. This means that like Screens, which expose Activities, a Task can be customized by an App Inventor user. A user will be able to add multiple Components to a Task and design behavior for the Task by defining actions for these Components and specifying interactions between Components.

3.2 INTENTS

Before delving into Android Services, let us take a step back to understand Android Intents. An Intent in Android is a data type that is essentially a description of an action to be performed [3]. It contains details of the type of action to be performed and holds any information that might be relevant to that action. The following are some situations in which Intents are used: to start activities, to start services, and to stop services.

3.3 Android Services

As defined in the Android Developer’s Handbook, “a Service is an application component that can perform long-running operations in the background and does not provide a user interface.” [2] There are many types of Services in Android, and all were considered when deciding what to use for the App Inventor Task implementation.

1. Started Service

A Started Service usually performs one particular operation and then kills itself. It needs to be explicitly started by another application component. The
3.3. ANDROID SERVICES

CHAPTER 3. BACKGROUND TASKS

Started Service continues to run and execute its tasks even when the component that started it has stopped. These are started by calling the startService() command and passing an Intent which has any necessary information for the service. The startService() command calls the onStartCommand callback which is where the details of the Service’s tasks are implemented.

2. IntentService

The IntentService class helps simplify the Service implementation for a situation in which multiple requests are not handled simultaneously. The IntentService class has a default implementation of the onStartCommand callback which forwards calls one at a time to the onHandleIntent callback. Therefore Intents get processed one at a time by the onHandleIntent callback thereby avoiding multi-threading issues. The only thing that needs to be implemented is the onHandleIntent callback.

3. Bound Service

Other application components “bind” to a Bound Service. A Bound Service exists and performs operations only as long as some component is bound to it. The components bound to this type of Service and the Service interact with each other through interprocess communication (IPC) \[4\]. Application components bind to this type of service by calling bindService. This calls the Service’s onBind callback which returns an IBinder object. An IBinder basically exposes functions that the binding component can then call. The components can then interact with the Service through this IBinder object that they both share.
3.4 Considerations for AI2

I call my abstraction for Services a Task. A Task runs user-specified actions in the background and provides no user interface. Therefore this Task abstraction seemed apt.

App Inventor is designed to be a platform that can be used even by novices. As such, it is important to keep the platform very intuitive. Therefore, when deciding which type of Service to use, I considered what would maintain the simplest abstraction.

I ruled out Bound Services, because they put too much heavy lifting on the App Inventor user. As described in Section 3.3, to use Bound Services, one must implement the IBinder object which is returned in the services onBind callback when a component binds to it. An IBinder defines an interface through which components can interact with the Service. An example IBinder is shown in Figure ( ). Methods and Events are mainly how functionality is implemented in App Inventor. And introducing this new notion of communicating through a user-defined interface complicated this simple platform. Other types of Services fit better into the Methods and Events framework.

The only difference between an IntentService and a regular Service is that an IntentService processes Intents one at a time while a regular Service allows for the possibility of simultaneous processing of events. I chose to use the regular Service class in order to allow future members of the App Inventor team to easily extend support for this simultaneous execution. Although a Task abstract the regular Service class, my implementation is such that currently the behavior is like that of an IntentService. I think this is the best approach as we first introduce Services to the platform, so users are not tasked with addressing difficult multi-threading problems. However, since the Task class extends the Android Service class and not the IntentService class, it can easily be extended so that a user-set flag determines
whether to allow simultaneous processing or not.

Android allows Services to be run in a separate process. However, doing so makes communication between application components and the Service much harder as we must deal with inter-process communication. The only time this is really useful is when a custom Service needs to be accessed by other applications. So, for the sake of maintaining simplicity in App Inventor, this first iteration of Tasks does not support use by external applications.

3.5 Top-Level Components

Because a Task is a top-level component, the changes made span various parts of the App Inventor codebase. Simply adding a regular component is not as complicated. However top-level components must be able to contain other components. Like a Screen, other top-level components must be associated with their own Designer view and Blocks view in the AI2 web application. During compilation, top-level components need to be declared in the AndroidManifest file. Top-level components have their own environments in the Kawa runtime which we will discuss in detail later.

3.6 Tasks Interface

Before we explore the details of the implementation of Tasks, let’s define what our end goal is. We want to give users the ability to use Tasks in their applications. How will users interact with App Inventor to make this happen? The Screen class needs to be able to start and stop Tasks. There are two methods StartTask and StopTask that allow this functionality. Section 3.9 looks at these methods in detail.

The user also needs to specify behavior for the Task to execute when it is started.
3.7. TASK IMPLEMENTATION

The TaskStarted event is triggered when a Task is started and a user can specify behavior there. Section 3.7 goes into detail on the Task

In the big picture, a Screen starts a Task and then the user-specified behavior in the TaskStarted event is executed. In the following sections, we delve into what goes on behind the scenes to provide this simple interface for the user.

3.7 Task Implementation

Each Screen in an AI2(App Inventor 2) application is represented by a Form class that extends the Android Activity class. Similarly each Task is represented by a Task class which extends the Android Service class. As we now know, when a Service is started the onStartCommand callback is evoked. When a Task is started, the onStartCommand callback launches the Task’s TaskStarted event. The App Inventor user can then specify what happens when the TaskStarted event is received.
Other important events are Initialize which is launched when the Service class is first created and Destroy when the Service class is destroyed.

![Diagram of Task events]

Figure 3.3: Example of user-defined behavior for Task events

The Task class also implements the Component interface, the ComponentContainer interface, and the HandlesEventDispatching interface. The Component interface implements a getDispatchDelegate function which keeps track of the top-level component which contains it. For a Task, this function returns a reference to itself.

The ComponentContainer class is implemented by every Component that can contain other Components. It keeps track of the parent Component. For a Task, the parent component is itself. The ComponentContainer interface has a $form function and a $task function. In a Form, the $form function returns the Form and the $task function returns null. For a Task, the $form function returns null and the $task function returns the Task itself.

The HandlesEventDispatching interface is implemented by Forms and Tasks. This allows Forms and Tasks to define events which will be launched when the Android application executes. App Inventor users can then specify actions that occur when these events are launched. The HandlesEventDispatching interface acts as the bridge between these user-defined actions and the Android runtime which launches these events.
3.8 Web Application

3.8.1 File Editors

As described earlier, different File Editors represent the user designed-blocks and designer files in the web application. Previously any top-level component in an application was a Screen, and therefore there were only two types of editors a FormEditor and a BlocksEditor. With the introduction of Tasks, I needed to create a new type of FileEditor, the TaskEditor. The same BlocksEditor could be used to represent the blocks code for a Task. So every Form will be associated with one FormEditor and one BlocksEditor and every Task will be associated with one TaskEditor and one BlocksEditor.

![Task Designer view and its relation to the TaskEditor components](image)

Figure 3.4: Task Designer view and its relation to the TaskEditor components

Each Task designer view is made up of a NonVisibleComponentPanel which users drag child components onto. While a Screen contains both a VisibleComponentPanel and a NonVisibleComponentPanel, the Task has no user interface and therefore has no VisibleComponentPanel. There is a PropertiesPanel which is where users modify component properties and there is a PalettePanel which displays all the components. A MockTask object describes how to render the Task component in the Designer
View. The TaskEditor keeps track of all these UI elements.

The TaskEditor also has event handlers which handle changes the user makes to the designer view. The names of these event handlers is quite self-explanatory: onComponentPropertyChanged, onComponentAdded, onComponentRemoved, onComponentRenamed, onComponentSelectionChange. These become especially relevant when live-testing on the App Inventor Companion application. These handlers update the structure of the application and then push these updates to the phone. The BlocksEditor has similar handlers which push changes to the phone.

The TaskEditor also handles JSON encoding of the Task structure. This json representation is used in later stages when the application is actually compiled.

### 3.8.2 Adding Tasks

Now let’s look at what overhead work needs to happen when a Task is first created. When a Task is created using the “Add Task” button a dialog pops up showing you an automatically suggested name for your Task. For example, if you have added two tasks named Task1 and Task2, the next auto-generated name will be Task3. You can also name the Task whatever you want. This new Task name is then added to the dropdown of the AI2 application’s top-level components. Previously, this was just a dropdown of Screens. I decided to show Screens and Tasks in the same dropdown since users will only work on one Designer view at a time either way.

So when a Task is actually added, the source files for the task file(.tsk) and the blocks file(.bky) are created. The task file is then associated with a TaskEditor and the blocks file is associated with a BlocksEditor. A Task object is created which keeps track of the TaskEditor and BlocksEditor associated with that Task. The item in the dropdown has a reference to this Task object. So when a user attempts to switch to this Task view, the TaskEditor renders the Designer view appropriately. The BlocksEditor does the same for that Task’s Blocks view.
3.9. FORM CHANGES

CHAPTER 3. BACKGROUND TASKS

3.8.3 Miscellaneous Changes

Because App Inventor previously only supported Screens, many parts of the application assumed that any top-level container was a Screen. Many functions expected a Form, MockForm or FormEditor as input. App Inventor also uses file extensions to determine what files represent. "bky" files are blockly files that represent the blocks associated with a top-level component. "scm" contain the JSON representation of a Screen and its components.

Tasks are now represented with the "tsk" representation. Even though the contents of "scm" and "tsk" files are similar in representation, because file extensions are indicative of container type it was important to create a new file extension.

The SimpleEditor class is now a parent class to the FormEditor and TaskEditor. Functions that previously expected FormEditors, expect SimpleEditors. The MockTask and MockForm classes now extend MockTopLevelContainer. Previously the containing context of any component was an Activity, now it can be either an Activity or Service and the context needs to be cast accordingly. Similar changes were made throughout the platform, to support Screens and Tasks simultaneously.

3.9 Form Changes

As discussed, Services are started when other application components call startService. What are these other components? In Android, a Service can be started by an Activity or another Service.

Since Activities can start services, the capabilities of an App Inventor Screen need to be updated now that Tasks are involved. Forms can now start and stop services. In Android, a user can use Intents to pass values between components when an action is being performed. For example, when a Screen starts a new Screen, the App Inventor user can pass a value with that Intent to start a form.
These values can then be used to initialize the Screen or perform any pre-processing that is necessary.

In Android, these values can be of any type. The Intent then gets serialized and developers specify the type of each value when reading values from the Intent. To keep things simple, I decided to support the passing of a single String value when starting a service. This String value can itself represent multiple values which can be parsed and used on receipt. Since this is similar to what happens when a new Screen is started, this fits well in the App Inventor platform. I will go into more detail about other approaches that could have been taken and why this is the best when discussing the ApplicationMessenger Component which has a similar need to pass values with an Intent.

For now, the App Inventor Screen class has three new methods: StartTask, StartTaskWithValue and StopTask. The functions of these follow directly from the names. StartTask and StartTaskWithValue both call the Android Activity’s startService function with an Intent representing the Task that is to be started. For StartTask, the Intent’s serviceArg value is set to an empty string and for the StartTaskWithValue the Intent’s serviceArg value is set to the string passed in by the user.

### 3.10 Blocks to App

YAIL is the intermediate language that App Inventor uses to compile blocks to a packaged Android application. YAIL is basically a small set of functions and macros defined in kawa. So, blocks are translated to YAIL, the YAIL hooks into the kawa runtime which then creates the appropriate Java classes to represent the Screens, Tasks and child Components in an application.
3.10. BLOCKS TO APP

CHAPTER 3. BACKGROUND TASKS

(a) Starting a Task from a Screen

(b) Starting a Task and passing a value

(c) Stopping a Task

Figure 3.5: Example of starting and stopping a Task

Figure 3.6: Example YAIL code to add a component and define an event handler for that component

3.10.1 Generating YAIL

As in other parts of App Inventor, here too, App Inventor formerly assumed that any file it received was for a Screen. Screens are represented by the Form Java class and Tasks are represented by the Task Java class. Therefore, the YAIL generated needs to clearly indicate this distinction. I added two separate methods getFormYail and getTaskYail to generate YAIL for the different top-level components appropriately. getFormYail generates the YAIL for Screens as was previously done.

getTaskYail takes in the json representation of a Task. This has information about the Properties of the Task as set in the Properties Panel, the Components
that are a part of the Task, and the properties of those Components as well.

The first line of YAIL code for a Task is a call to define-task — define-task is described in the next section. After this, the YAIL code generated references the blocks that were added in the blocks view of the Web application. Each block in App Inventor knows how to generate its own YAIL code.

First, getTaskYail loops through all the global blocks which are not a part of any Component and asks each block to generate its YAIL representation. These global blocks could be any global variables or methods that are defined by the user. Next, from the json representation, YAIL code is generated to set the properties of the Task. Then, we generate YAIL for all the top-level blocks i.e. the blocks based on the parent Task component. This could be any methods of the Task that are called or any events that the user defines actions for. Finally, we loop through all the child Components recursively and generate the YAIL for each — to set the properties described in the json and for any Component blocks like methods and events.

### 3.10.2 Tasks in Kawa

As described earlier, the YAIL code hooks into the Kawa bridge. The YAIL for a Task calls the define-task method in Kawa to create the associated Task java class and perform all the overhead actions necessary for the app to run smoothly.

Each Task has an Environment in the Kawa runtime which can store key-value pairs. There is an add-to-form-environment method which adds components, methods, events and variables to the Task’s Environment. The lookup-in-task-environment method finds components, variables, etc. in the Task’s Environment. There is a global-var-environment associated with each Task which stores all the global variables used in the Task. There is a list of events, events-to-register, to register for this Task. There is another list of components, components-to-create.

When a Task is first initialized, all the events in events-to-register get registered
with the EventDispatcher. The EventDispatcher is used in the Java implementation of the various components to launch events of the appropriate type. The global variables are then initialized and stored in the global-var-environment. All the Components listed in components-to-create are created; this means their Java classes are created. References to these component objects are then added to the Task’s Environment. In the future, when methods or variables of the Component need to be accessed, lookup-in-task-environment will use this reference. All the Components initial properties are set as described in the Properties Panel on the Web application.

In define-task we also define a Task’s behavior when events are received. This is done in the dispatchEvent method. When the EventDispatcher dispatches a particular event, the Java class hooks into the dispatchEvent method defined here. When an event is received, the dispatchEvent method looks up the appropriate handler for that event using lookup-in-task. It then passes the correct arguments to the handler and runs the user-defined actions.

Error handling behavior is also defined in the Kawa code. Screens and Tasks differ in their error handling behavior. Because Screens take up foreground space they can report errors through Toasts and Dialogs which also take up screen real estate. However, Tasks do not have the same luxury. Tasks can report errors through Android log messages which can be seen in the logcat viewer and through notifications. Since a notification doesn’t take up foreground real estate unless the user pulls down the notifications drop down, Tasks can also launch notifications.

### 3.10.3 Kawa Runtime Changes

When the only top-level components involved were Screens, App Inventor applications were much easier to rationalize. An application was made up of multiple Screens and only one Screen can be in the foreground at a time. Therefore, as long as we had a reference to the current Screen all lookups for Components, global
variables, functions etc could look in the Environment associated with that Screen.

However, with Tasks in the picture, lookup operations need to know whether to look in the Task’s Environment or the Screen’s Environment. Moreover, multiple Tasks can run at the same time, since there is no notion of a single foreground Service in Android. For simplicity’s sake, we will start out with only one Task per application. So the Kawa runtime will have a reference to the current Screen as well as the single Task — if it has been created — allowed for an application.

Now, the methods that perform the lookups need to know whether the lookups should happen in the Screen or Task Environment. As such, the method signature of these has been updated to take in context as a parameter. The context is a String whose value can either be “Screen” or “Task”. The functions which have been modified are:

- **lookup-in-current-environment** This replaces the previously used lookup-in-current-form-environment method. There is a new lookup-in-task-environment method. The lookup-in-current-method wraps these two other functions and returns what is appropriate based on the context. These functions lookup and return values from the appropriate environments.

- **add-global-var-to-environment** This replaces the old add-global-var-to-current-form-environment. There is a new add-global-car-to-task-environment. The add-global-var-to-environment is a wrapper for these which adds the global variable to the correct environment.

- **lookup-global-var-in-environment** This method is also a wrapper which looks up global variables in the appropriate environments based on the context given.

Many other methods which use these modified functions were also modified to take context as a parameter. Some examples are: get-component, get-property, get-
var, set-var!, call-component-method. The names of these functions are indicative of their behavior.

3.10.4 Block Context

In Section 3.10.1, we learned that blocks know how to generate their representative YAIL. When the only top-level components in App Inventor were Screens, there was a notion of the “current” Screen, since only one Screen could be in the foreground at a time. So when the YAIL for blocks was being generated, there was no ambiguity as to which component or variable was being referenced. For example, if there is a block referencing a global variable, I know that this global variable is associated with the current Screen.

However, with the addition of Tasks, there are multiple things that are active at the same time. So if a global variable is being referenced, it now becomes important to distinguish between global variables of the current Screen and global variables of an active Task. This information needs to be propogated to the Kawa runtime, so that lookups happen in the correct Kawa environment.

This means the YAIL needs to get this information from the blocks and then pass this on to the Kawa runtime. The blocks themselves need to be aware of their context – whether they are in a Screen or a Task. For this, I added a context field to the block data structure. When the YAIL is being generated in getFormTask and getYailTask — these functions were discussed in Section 3.10.1 — a block’s context gets set. When the blockToCode, statementToCode, and valueToCode functions get called to generate a block’s code, the context is now passed on to the child blocks. For example, an event block passes its context on to the action blocks contained within it. This way, every block in an application knows which type of top-level component contains it.
3.10.5 Android Manifest

A simple but important change to the Android Manifest file needs to be made for the compiled application with Tasks to work. Each Task added to the application needs to be explicitly declared in the Android Manifest. This is done in Compiler.java.
Chapter 4
ApplicationMessenger

Tasks allow developers to design applications that perform operations in the background. However, a user’s main interface to interact with an application is still through the Screens of an application. Therefore, it is important to somehow enable Tasks and Screens to communicate with each other. The ApplicationMessenger Component performs this role. The ApplicationMessenger is an abstraction of the Android LocalBroadcastManager.

4.1 Broadcast Receivers

A Broadcast Receiver is an Android component which can register for certain events — sent by the system, other applications, or the parent application itself. When a certain event occurs, Android notifies all the registered receivers [5]. These receivers can then react to these events accordingly.

What are these “events” that a Broadcast Receiver can register for? In fact, these “events” are just Intents. An Activity or Service can broadcast an intent through the sendBroadcast(Intent intent) function. The Intent that is broadcast can hold any information that the broadcasting Activity or Screen thinks is relevant. For example, a Service whose job is to download a file can send a broadcast when it has finished downloading. Some relevant information for this event could be the file type.
of the downloaded file, the size of the downloaded file, and the location of the downloaded file. So the Service can create an Intent, add all this relevant information to this Intent, and then broadcast the intent through the sendBroadcast function. For broadcasts it is important to set an action for the Intent. Any interested BroadcastReceiver will register for specifically for that action. So this example Service would assign the action “file-loaded” to the Intent it broadcasts. Any BroadcastReceivers who register for the “file-loaded” action will then be notified of this event when the Intent with that specific action is broadcast.

4.2 Security

As listed in the Android developer documentation, there are some security considerations to keep in mind when using broadcasts [5].

1. The intent name space is global. This means that it is important to assign actions to events that are unique enough that they don’t conflict with the intents broadcast by other applications.

2. Any applications can send broadcasts to a particular receiver.

However, these security considerations can be avoided by using the LocalBroadcastManager. Intents broadcast through the LocalBroadcastManager never go outside the application. To keep things simple for the App Inventor user, I decided to use the LocalBroadcastManager instead of the generalized BroadcastReceiver. There are many advantages to using a LocalBroadcastManager

1. We avoid security holes that can be exploited because of the problems listed above.
2. Users will not have to reason about permissions and who is allowed to see which broadcasts and so on. This is important since simplicity is an important characteristic of the App Inventor ecosystem.

3. It is more efficient to send broadcasts using the LocalBroadcastManager than sending global broadcasts.

4.3 Specifications

Let’s take a look at the methods, events, and properties of the ApplicationMessenger component.

4.3.1 Properties

The ApplicationMessenger has only one property, LocalMessagesToReceive. The LocalMessagesToReceive property is a comma-separated list of the message types that the ApplicationMessenger will respond to.

![ApplicationMessenger Properties](image)

Figure 4.1: ApplicationMessenger Properties

4.3.2 Methods

The ApplicationMessenger has a single method, SendLocalMessage. This method takes as arguments the type of message being sent, messageName, and the value to actually pass along, message. SendLocalMessage then broadcasts this message for anyone listening.
4.3.3 Events

There are five events that can be triggered in the ApplicationMessenger. These are LocalMessageReceived1, LocalMessageReceived2, LocalMessageReceived3, LocalMessageReceived4, and LocalMessageReceived5. The five message types specified in the the LocalMessagesToReceive property are handled by these five events.

4.4 Implementation

In this section, we look in detail at the changes made to support the LocalBroadcastManager in App Inventor.
4.4. IMPLEMENTATION

4.4.1 Android Support Libraries

To use the LocalBroadcastManager we need to use the Android Support Libraries. The Support Libraries provide backwards-compatible versions of many of the Android framework APIs. The LocalBroadcastManager is in fact only available through the Support Libraries. So, I added the support libraries to the App Inventor platform.

4.4.2 Defining Messages

The ApplicationMessenger needs to allow App Inventor users to do the following:

- Define events to register for.
- Specify behavior for each of the event types that can occur.
- Send broadcasts of a specific type and attach any relevant information.

This brings up an interesting problem. In App Inventor, there is currently no way to dynamically generate blocks. For example, if a user defines an event “file-loaded”, we want an event handler for this message i.e. a when-block that the user can use to define behavior when the event is received. Ideally, whenever the user defines a new message type, we generate a new when-block for the ApplicationMessenger component. So not every ApplicationMessenger would have the same when-blocks.

However, this is not feasible with the current App Inventor setup. Because, setting up such a system is beyond the scope of my thesis, I decided to work within the existing limitations. To re-state, the problem is that if App Inventor doesn’t know what these event types are beforehand, there is no way to generate the when-blocks ahead of time and App Inventor does not support the dynamic generation of blocks.

To handle this problem, I decided to limit the number of event types that an ApplicationMessenger can register for. Each ApplicationMessenger can register upto
five event types. This is done by entering the event types as a comma-separated string in the LocalMessagesToReceive property. If a Screen needs to listen to three event types, “downloaded”, “uploaded”, and “saved”, the LocalMessagesToReceive property would be “downloaded, uploaded, saved”. There are five message handlers in each ApplicationMessenger: LocalMessageReceived1, LocalMessageReceived2, and so on. The order of the event types in the LocalMessagesToReceive property directly corresponds to the order of the event handlers. So in the above example, the “downloaded” event is handled by LocalMessageReceived1, the “uploaded” event by LocalMessageReceived2, and the “saved” event by LocalMessageReceived3.

4.4.3 Receiving Messages

When the LocalMessagesToReceive property is set, a BroadcastReceiver is created for each of the messages. The onReceive method of the BroadcastReceiver is called whenever the message it has registered for is triggered. The onReceive method then triggers the corresponding event handler. For the same example used in the previous section, the BroadcastReceiver for the “downloaded” event dispatches the LocalMessageReceived1 event. The user described behavior for this event is then executed.

It is important that once the BroadcastReceivers have been created, they are registered appropriately. Whenever the LocalMessagesToReceive property is updated, all the receivers are created and also registerd. We register the receivers by calling LocalBroadcastManager.registerReceiver(BroadcastReceiver receiver, IntentFilter filter). The BroadcastReceiver is the receiver to be registered and the IntentFilter specifies the action that this receiver should respond to.

In Screens, it’s important to also unregister the receivers when the Screen is not in the foreground. So we register receivers in the onResume method, which is called when a Screen comes to the foreground. We unregister listeners in the onPause
method, which is called when the Screen moves out of the foreground. Whenever the LocalMessagesToReceive property is updated, it is important to unregister all existing receivers before registering the new ones. Otherwise, multiple receivers will be attached to the same event and the event handlers will be called multiple times.

4.4.4 Sending Messages

The SendLocalMessage method of the ApplicationMessenger takes a message name and message contents and sends a broadcast. To send messages, we use the LocalBroadcastMessenger.sendBroadcast(Intent intent). The message contents are added to the intent as a string by calling intent.putExtra and assigning a name to the message contents. When messages are received, the message contents are retrieved by calling intent.getStringExtra and specifying the name that was assigned to the message. This information is then propagated to the event handlers.
Chapter 5

Notifications Component

The Notifications component is another important part of the Tasks ecosystem. Because Tasks don’t have an interface of their own, the Notifications component can be used to launch a notification containing important information that needs to be conveyed to the user. Another way to communicate with the user is to send information to the current Screen which can then display the information somehow. However, this becomes tricky because:

1. The foreground Screen can change. To ensure that any Task’s message is conveyed to the app user, the App Inventor user would have to set up appropriate message handlers on every single Screen in the application. This way no matter what the foreground Screen is, the Task’s message is handled.

2. There is no simple way for a Task to convey a message to the user if the parent application is not in the foreground. Since Tasks are not tied to Screens, it is very possible for a Task to be running even when a Screen of the application is not in the foreground.

3. The Task’s message may not be relevant to the foreground Screen. For example, if the Task finished downloading a file, it might be a better user experience for the status message to be delivered using a notification since the foreground Screen might not find that information relevant.
5.1 Background

The App Inventor notifications component allows users to launch a notification with a specific notification id, title, and text. A notification id helps identify the notification to update and delete later. Launching another notification with the same id as an existing notification causes the existing notification to update instead of launching a completely new one.

Notifications in Android depend on two key Android classes, the NotificationCompat.Builder and the NotificationManager. The Android NotificationManager handles launching, updating and cancelling notifications. The NotificationCompat.Builder handles designing and constructing a notification as per user specifications. So the Builder designs the UI of the notification and the NotificationManager handles launching and deleting of the notification.

The original Notification.Builder class was introduced in API Level 11. However, the Android Support Libraries contain the NotificationCompat.Builder class which allows apps to support notification on versions of Android as old as API Level 4. The Android Support Library provides backward-compatible versions of the Android framework APIs targeted at a specific API level [6]. For the notifications component, I used the Support Library for API Level 4. Because the ApplicationMessenger Component also used the Android Support Libraries, I did not have to add any new libraries the App Inventor platform. The necessary changes were already made.

App Inventor’s minimum supported Android level is API Level 3. However, using the Android Support Libraries, which are necessary to access NotificationCompat.Builder, requires the minimum sdk to be API Level 4 at the least. Increasing the minimum sdk level causes problems with the sizing of different App Inventor components. This problem is being addressed by a separate project in App Inventor.
5.2 Specifications

The NotificationComponent does not have any properties or trigger any events. Its core functionality is wrapped into four important methods.

5.2.1 Methods

1. **LaunchNotification**
   
   This method launches a notification with a specific title and text. A notification id is specified so that the notification can be deleted or updated in the future.

2. **LaunchNotificationWithRedirect**
   
   This method like LaunchNotification launches a notification with specified
5.3. IMPLEMENTATION  CHAPTER 5. NOTIFICATIONS COMPONENT

information. However, this method can be used to redirect the user to a particular Screen when a notification is clicked. Notifications launched using LaunchNotification don’t redirect the user to any Screen.

3. CancelNotification
This cancels a notification with a specific id.

4. CancelAllNotifications
This cancels all notifications launched by this component.

5.3 Implementation

The Notifications component is a non-visible component. It does not have any properties like the “hour” and “minute” properties of the Time Picker component, or any events. However, it has many methods that allow users to launch and delete notifications as necessary.

Each instance of the Notification Component has its own NotificationManager. The NotificationManager hooks into the Android system service that handles notifications.

The LaunchNotification methods takes as input a notification id, title and text. The NotificationCompat.Builder, from the Android Support Libraries, is used to construct this new notification. For now, I use the default Android icon as the notification’s icon. This is because notifications need to be compatible with Tasks and the compatibility of assets and Tasks is still being investigated. However, once Tasks and assets are compatible, the notification icon can easily be made customizable. When a notification is launched, a PendingIntent can be associated with it. The PendingIntent defines the behavior of a notification when it is clicked. The PendingIntent can point to a Screen in the application which is brought to the
foreground when the notification is clicked. LaunchNotificationWithRedirect takes a screenName and value as inputs. When a notification is launched using LaunchNotificationWithRedirect, the notification’s PendingIntent directs the user to the screen specified by screenName and passes the specified value as the start value of the Screen. When launched using LaunchNotification, notifications act as simple informative messages and do not redirect to a Screen on user interaction.

The CancelNotification method passes the specified notification id to the Component’s NotificationManager and instructs it to delete that notification. The CancelAllNotifications method instructs the NotificationManager to cancel all notifications that it launched itself.
Chapter 6

Base Task Component

App Inventor previously supported two types of Components: Visible Components and Non-Visible Components. You can probably guess the distinction. Visible Components are Components like a Button, Image, Label, etc. These Components take up space on their parent Screen. On the other hand, Non-Visible Components like the AccelerometerSensor, LocationSensor, OrientationSensor, etc. are not physical UI elements. They perform important operations but do not actually take up any space on their parent Screen.

However, in practice, the distinction is not actually that straightforward. There are some Components which do not initially take up space on the parent Screen but they later launch their own Screen to perform certain functions. For example, let’s take a look at the Camera Component. The Camera Component is categorized as a Non-Visible Component. This makes sense because adding a Camera Component to a Screen does not actually embed a camera view on the Screen. However, when the user wants to take a picture, the Camera Component opens a new pre-built Camera Screen for the user to interact with. So though the Camera is a Non-Visible Component, it does modify the UI of the application.

These “in-between” Components, like the Camcorder and the Camera, present a problem for Tasks. Tasks are non-visible top-level Components in App Inventor. Tasks do not have a user interface; they perform their operations in the back-
ground. So while Screens can contain both Visible and Non-Visible Components, Tasks can only contain Non-Visible Components. However, can Tasks contain these “in-between” Components? No, they cannot. The Components in a Task cannot ever modify the user interface of an application during their lifetime. To solve the problem of these “in-between” Components, I created another type of Component, the Task Component, represented by the new NonVisibleTaskComponent class. The Task Components are non-visible components which can function in both Screens and Tasks.

6.1 Task Component

The NonVisibleTaskComponent Java class is the implementation of the Task Component. All Components that can be contained in a Task will extend this class. Although this Component type is called the Task Component, it is important to realize that Task Components can actually also be added to Screens. However, to avoid confusion with the regular Non-Visible Components, which cannot be contained in Tasks, we call the non-visible Components that can be contained in Tasks “Task Components”.

Depending on whether the parent Component of a Task Component is a Screen or a Task, the actual logic performed may be different. For example, the ApplicationMessenger Component described in Chapter 4 behaves differently in Screens and Tasks. In Screens, the Broadcast Receivers are unregistered in a Screen’s onPause callback and are registered in the onResume callback. In Tasks, the receivers are unregistered in the onDestroy callback but nowhere else.

To allow this, the constructor of NonVisibleTaskComponent takes in an instance of ComponentContainer – this class is implemented by any Component that can contain another Component e.g. Screens and Tasks. ComponentContainers have
$form and $task attributes which indicate what the top-level parent Component is. The NonVisibleTaskComponent stores what this parent component is in the form and task attributes.

The NonVisibleTaskComponent implements the Component interface and must therefore implement the getDispatchDelegate and dispatchErrorOccurredEvent methods. getDispatchDelegate returns the top-level Component which will handle dispatching events. If a Screen is the top-level Component, the form variable is returned. If a Task is the top-level Component, the task variable is returned. The dispatchErrorOccurredEvent method simply forwards the call along to the containing form or task, whichever is relevant.

### 6.2 New Task Components

Converting a Component to a Task Component requires the Component to behave smoothly in both Tasks and Screens. This means the Component developer may have to implement slightly different behavior based on the type of the parent Component. I have converted a few of the existing Non-Visible Components to Task Components.

1. **ApplicationMessenger**

   In the ApplicationMessenger Component, if the parent container is a Screen we register for the onPause and onResume listeners. If the parent container is a Task we register for the onDestroy listener. The onPause, onResume, and onDestroy listeners are lifecycle methods that are triggered by the Android framework. The onPause callback is triggered when a Screen moves out of the foreground. The onResume callback is triggered when a Screen moves into the foreground. The onDestroy callback is triggered in both Screens and Tasks when they are destroyed. So in the constructor, we check whether the
container is a Screen or a Task and register for the appropriate callbacks. All three callbacks are implemented — the behavior of each callback was described earlier in this chapter. Only the callbacks that are registered for get executed.

2. **Notification**

Each notification that is launched by the Notification Component is associated with a PendingIntent. This PendingIntent tells the notification which Screen to redirect to when the user eventually clicks on the notification. If a notification is launched from a Screen, the PendingIntent redirects to the Screen that initially launched the notification. If a notification is launched from a Task, the PendingIntent does not redirect to any Screen. Therefore, in Tasks the notifications act as more of an informative tool than a gateway back into the application.

3. **Clock**

The changes made to the Clock Component are also based on the lifecycle callback methods — similar to the ApplicationMessenger. Most functionality in the Clock Component like FormatDate, FormatTime, and DayOfMonth is independent of the containing Component type. However, the Timer feature specifically relies on the lifecycle callback methods to be implemented properly. The App Inventor user sets the TimerInterval of the Clock Component which causes the Timer event to be triggered after every interval. However, in Screens the Timer event is triggered only if the Screen is in the foreground. This is because, if the Screen is not in the foreground and the user makes UI changes, an error will be thrown. So the onResume and onStop lifecycle callbacks help the Timer keep track of whether the Screen is in the foreground or not. For Tasks, there is no notion of being in the foreground and therefore the Timer event is always triggered until the Task is destroyed. In both Screens and Tasks, the Timer is disabled when the component is destroyed. This ensures
that the Timer does not try to trigger events on a non-existing Component.

4. **File**

Unlike the Components we’ve looked at so far, the File Component behaves the same in both Screens and Tasks. The change made here was not to implement different behavior for Screens and Tasks. Previously, the File Component called `activity.runOnUiThread` to trigger events on a Screen from another thread. The `runOnUiThread` method is only available to Android Activities and therefore App Inventor Screens. To make this Component compatible with Tasks, I replaced all calls to `runOnUiThread` with the `Handler.post` method which is compatible with both Screens and Tasks.

5. **Web**

Just like the File Component, the Web Component behaves the same in both Screens and Tasks. The only change that had to be made was to use `Handler.post` instead of `activity.runOnUiThread` to dispatch events.

### 6.3 General Guidelines

So, the Task Component lays down the groundwork for future development of Components for Tasks. Some general guidelines for Component developers are:

1. To adhere to App Inventor’s principal of simplicity, it’s important that any differences between a Component’s behavior in Tasks and in Screens be abstracted away from the App Inventor user. It should not be the user’s responsibility to discern which methods to use in Tasks and which to use in Screens. The changes should be behind the scenes.

2. It is important to keep in mind the lifecycle of Screens and Tasks and their
callback methods. The ApplicationMessenger and Clock Components are good examples of this.

3. To dispatch events in Task Components it’s important to use Handler’s which are compatible with Tasks and Screens instead of Activity.runOnUiThread which can be used from Visible Components.
Chapter 7
Sample Applications

Now that we have a good understanding of the Tasks ecosystem and its capabilities, let’s take a look at some sample applications that use Tasks. These sample applications will help put in perspective what the previous limitations of App Inventor were and how Tasks address some of these issues.

7.1 File Application

First, we will take a look at a simple file downloading application. The Task downloads a file in the background and notifies the Screen when it’s done. The Screen then loads that image. This is useful for applications like Pinterest and Facebook, which display many images at a time.

1. We add a Button component and Image component to Screen1. We also add an ApplicationMessenger component which can be found in the Sensors category.

2. Create a task, Task1, and add a Web Component and an ApplicationMessenger component. In the properties panel we set the url to http://upload.wikimedia.org/wikipedia/commons/1/17/Tiger_in_Ranthambhore.jpg or the url of any image. We check that the SaveResponse toggle is enabled and type in a name for the ResponseFileName like “response.jpg”.

55
3. When the Task is started, the Web component issues a get request for the picture. When the file is received, we broadcast a message. The name of the message is “loaded” and the message contains the file name of the response.

4. In the Screen1 Designer View, set the LocalMessagesToReceive property of the ApplicationMessenger component to “loaded”. The ApplicationMessenger component can listen to a maximum of five different messages. The LocalMessagesToReceive property is a comma-separated list of the five message types.
5. When the Button is clicked, we start the Task. When we receive the “loaded” message from the service, we load the saved image into the Image component. The Screen only receives messages when it is in the foreground. A TinyDB can be used to communicate between a Screen and a Task even when the Screen is not in the foreground.

7.2 Nearby Application

Next, we will look at how to build an application that recommends nearby restaurants. We will use the Google Places API for this.

1. First, in Screen1 we add a button and set up the blocks such that Task1 is launched when the button is clicked.

2. We create another Screen, Screen2 where the results will eventually be displayed. Let’s add a ListView to this Screen. We will come back to the blocks later.
3. Now, let’s create Task1. In Task1, we will use the LocationSensor, NotificationComponent and Web Component. We use the LocationSensor to detect changes in location. We use the Web Component to interact with the Places API and we use the NotificationComponent to notify the user that restaurants have been found nearby.

4. When the Task is started we enable the LocationSensor. You can configure the DistanceInterval and TimeInterval of the LocationSensor. For this sample application, we will leave the values at their defaults. The TimeInterval configuring how often location updates are received and the DistanceInterval configures what distance change(in meters) will trigger an update.

Figure 7.5: Button blocks for Screen1.

Figure 7.6: Task1 Initialize blocks.
5. When a location update is received, we use the Google Places API to find nearby restaurants using the lat/long coordinates of the new location. You need an API Key to use the Google Places API. You can easily obtain one by visiting the Google Developers Console and registering an application.

6. When we receive a response from the Web component, we use the Notification-Component to launch a notification and inform the user that nearby places have been found. We set up the notification with a redirect such that when the user clicks on it, Screen2 will be launched. So, the notification itself stores a PendingIntent for Screen2 that is launched when the user clicks on the notification. We pass the result from the Places API to the notification. When Screen2 is eventually launched, the result from the Places API is sent as the start value and can be accessed from Screen2.
7. So, whenever Screen2 is launched, we take the result from the Places API and display them in the List View. Because the Places API returns a JSON response we use the Web component to properly extract the relevant data.
Figure 7.11: Extracting information from response.
Chapter 8

Future Work

So far, we have gone over all that has been done to set up an ecosystem for Tasks in App Inventor. Now, let’s delve a little bit into the limitations of the system built and what further work can be done to expand the scope of Tasks.

8.1 Live Testing

A very convenient feature in App Inventor is the ability to live-test your applications. By downloading the App Inventor Companion application, users can sync the Companion application with their project on the web application. Changes made on the web application are reflected in the Companion application. This allows for real-time testing of applications. Unfortunately, this support has not yet been built out for Tasks.

8.2 Error Handling

Error handling on Tasks and Screens should be handled differently. For now, Screens popup a dialog with an error message while Tasks enter their error message into the Android log. The Task’s errors can be seen in Android’s logcat tool. However, it is possible to be a little creative with the error handling in Tasks. For example, if a
Screen of the application is in the foreground, the Task could pass its error message to the Screen using an ApplicationMessenger and then the Screen could display the message as usual in a dialog. The Task could also use notifications to deliver error messages to the user. Basically, there are many creative ways to deal with error handling in Tasks that can be explored. In the future, it would be beneficial to replace the simple error logging behavior of Tasks with a more user-friendly technique.

### 8.3 Blocks and Context

As described in Section 3.9, a block now has a context attribute which tracks whether its parent is a Form or a Task. Currently, a blocks context is determined when the Yail is being created. As the Yail is generated, the context is passed from the parent blocks on to child blocks. Instead, it would be convenient to determine the context upon creation of the block. This way, at the time of Yail generation, all blocks are already aware of their context. This seems like a better designed approach.

### 8.4 More Components

As described in Chapter 6, components need to be updated to be made compatible with Tasks. For now, I have updated a handful of components but there are many more in App Inventor that need to be made compatible. The Sound and YandexTranslate components, for example, would be very useful in Tasks.
Chapter 9

Conclusion

App Inventor is a great tool for novices to build Android Applications. It’s focus on simplicity really engages an audience that is otherwise hard to capture. At the same time, App Inventor is based on Android which is a constantly evolving platform. To make sure users get the best learning experience possible, it is important to make sure App Inventor evolves as Android evolves and to give users access to as much of the Android platform as possible.

Adding support for Tasks in App Inventor will allow users to create a variety of applications that could not be handled before. Besides the given sample applications, some interesting application ideas are:

1. An application that sends an email of your phone’s location every 15 minutes.

2. A custom music player.

With the NotificationComponent, ApplicationMessenger, and Tasks, users can now create much more powerful applications. The NotificationComponent allows apps to communicate with users even from Tasks. The ApplicationMessenger allows Screens and Tasks to communicate with each other. All together, the additions made in this thesis create a user-friendly environment for programming applications with Tasks. Meanwhile, App Inventor’s principle of simplicity has not been compromised.
Appendix A

Blocks to Yail

In Chapter 3, we learned that blocks are eventually translated to YAIL. Using the example from Chapter 1, we will take a look at the different stages of translation.

Figure A.1: Task1 designer view

Figure A.2: Task1 blocks

The components of a Task and its properties are encoded as JSON and then stored in a “.tsk” file.
The blocks of the Task are stored in xml representation in a “.bky” file.

```xml
<xml xmlns="http://www.w3.org/1999/xhtml">
  <block type="component" event_id="5" x="35" y="1">
    <mutation component_type="Task" instance_name="Task1" event_name="TaskStarted"></mutation>
    <field name="COMPONENT_SELECTOR" type="Task1"></field>
    <statement name="Do">
      <block type="component_method" id="19" inline="true">
        <mutation component_type="ApplicationMessenger" method_name="sendLocalMessage" is_generic="false" instance_name="ApplicationMessenger"></mutation>
        <field name="COMPONENT_SELECTOR" type="ApplicationMessenger"></field>
        <value name="ARG1">
          <block type="text" id="21">
            <field name="TEXT">Task1 has been started!</field>
          </block>
        </value>
      </block>
    </statement>
    <text>
      <mutation component_type="NotificationComponent" method_name="launchNotification" is_generic="false" instance_name="NotificationComponent"></mutation>
      <field name="COMPONENT_SELECTOR" type="NotificationComponent"></field>
      <value name="ARG1">
        <block type="text" id="27" inline="true">
          <field name="TEXT">Task1</field>
        </block>
      </value>
    </text>
  </block>
</xml>
```

Figure A.4: Task1 blocks in xml representation

The components, properties, and blocks are finally translated into YAIL. The YAIL representation is stored in a “.yail” file, shown in figure A.5.

This yail file is evaluated in the Kawa runtime and hooks into the Java code for the components.
Figure A.5: The yail for Task1
Appendix B

Updated Task Components

In Chapter 6, we learned that changes need to be made to components to make them compatible with Tasks. Let’s take a look at the changes made to the Web component to get a better understanding of these modifications.

Some of the changes are:

1. We create an instance of Handler in the Web constructor. The Handler allows us to run events on the UI thread from both Tasks and Screens. This is useful to dispatch events onto the UI thread.

2. Instead of keeping track of the parent Activity, we keep track of the parent Context. Both a Service and an Activity in the Android platform extend the Context class.

```java
private final Handler mHandler;
```

Figure B.1: Changes in Web constructor
APPENDIX B. UPDATED TASK COMPONENTS

3. When errors are dispatched, previously the component specifically called form.dispatchErrorOccurredEvent. Instead in Task components which extend the NonVisibleTaskComponent class, we call dispatchErrorOccurredEvent which then forwards the error to the Task or Form that contains the component.

4. Previously, to run an operation on the UI thread, components called the activity.runOnUiThread function. However, this function is not available to Services. Instead, we use the Handler instance that we created in the constructor. Previous calls to activity.runOnUiThread are replaced with calls to androidUIHandler.post where androidUIHandler is the instance of Handler.
Bibliography


