

Building Blocks for Mobile Games: A Multiplayer Framework for App Inventor for Android

by

Bill Magnuson

B.S., Massachusetts Institute of Technology (2009)

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 2010

Copyright Bill Magnuson 2010. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis
document in whole or in part.

Author.....
Department of Electrical Engineering and Computer Science
February 2, 2010

Certified by.....
Hal Abelson
Class of 1922 Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Building Blocks for Mobile Games: A Multiplayer Framework for App Inventor for Android

by

Bill Magnuson

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Building Blocks for Mobile Games is a client-server multiplayer game building framework for the App Inventor for Android platform. The framework includes an App Inventor component and a game server running on Google App Engine. The client side component packages the complexity of web service calls, data transfer and game state management into a set of graphical code blocks that allow users without programming experience to create Android applications that can access the game server API. The default game server provides basic functionality that can be used to create simple multiplayer games and message passing applications, such as a multi user bulletin board. By using custom server modules, the game server can be enhanced to provide server commands that implement a wide range of operations. Custom modules were used with the system to develop a multiplayer card game, a version of Bulls and Cows with a shared scoreboard, an application that accesses Amazon's book search API and a pair of applications for creating, managing and voting in polls. Using App Inventor and the game framework, each of these applications can be created with only a basic understanding of mobile phones and client-server principles.

Thesis Supervisor: Hal Abelson

Title: Class of 1922 Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I need to thank my advisor, professor, former boss and fellow Google intern Hal Abelson for starting this project, keeping me on track throughout the course of my thesis work and his incredibly valuable advice throughout the process.

I also owe extreme gratitude to the entire App Inventor team at Google. Special thanks to my manager and blocks editor hacking partner Sharon Perl, tech lead Mark Friedman, my always helpful desk neighbor, Liz Looney, the voice of reason in our cubicle, Ellen Spertus and my fellow interns Ben Gleitzman and Jill Dimond. Finally, thanks to Alfred Spector for all his help in securing my continued role on the App Inventor team during the writing of this thesis.

Additional thanks is due to Ben for going through all four years of MIT with me. While living together and bushwhacking through the new MIT computer science curriculum there was never a dull moment. It has been great.

I wish everyone involved and App Inventor itself the best of luck in the future. A lot of very intelligent people are hard at work on this project and I fully expect it to make waves in the years to come.

Contents

1	Introduction	14
1.1	App Inventor for Android	15
1.2	Games in App Inventor	16
1.3	Thesis Summary	17
2	Introduction to App Inventor	19
2.1	The Application Designer	19
2.1.1	Non-Visible Components	21
2.2	The Blocks Editor	23
2.2.1	Property Getters	25
2.2.2	Property Setters	25
2.2.3	Event Handlers	25
2.2.4	Method Calls	26
2.3	Framework Overview	26
2.4	Packaging and Running the Application	28
3	Building a Game	29
3.1	User Interfaces	30
3.2	The Voting Server Module	32
3.2.1	The Message Model	35
3.2.2	Input Validation and Error Handling	36
3.2.3	Registering Commands	37
3.3	Block Logic	38

3.3.1	Poll Your Friends Creator Blocks	38
3.3.2	Retrieving Polls and Casting Votes in Poll Your Friends	43
4	System Overview	46
4.1	The Game Server	46
4.1.1	Data Models	46
4.1.2	Request API	48
4.1.3	Extensions	52
4.1.4	Custom Modules	53
4.1.5	Game Server Testing	55
4.2	The Game Client Component	55
4.2.1	Properties	56
4.2.2	Method Calls	57
4.2.3	Events	58
5	Further Examples	61
5.1	Bulletin Board	62
5.2	MoBulls and Cows	64
5.3	Amazon	68
5.4	Androids to Androids	71
5.4.1	Round Numbers	72
5.4.2	Leaders	74
5.4.3	Messages and Persistent State	76
6	Related Research	79
6.1	Visual Programming Languages	79
6.2	Mobile Games in Education	81
6.3	Alternatives to Client-Server Design	83
7	Extensions	85
7.1	User Interfaces and Multiple Screens	85
7.2	Saving Local State	86

7.3	Pushing Messages and Game State Updates	86
8	Contributions	88
A	Game Server Code	89
A.1	Game Server	89
A.1.1	Models	121
A.1.2	Extensions	130
A.2	Custom Modules	145
A.2.1	Amazon	146
A.2.2	Androids to Androids	148
A.2.3	Bulls and Cows	155
A.2.4	Voting	158
B	Game Client Code	165
B.1	GameClient Component	165
B.2	Utilities and Data Structures	191

List of Figures

2-1	An empty application designer window. The application designer is just one part of the App Inventor web interface. The tabs visible across the top provide access to project management, sharing and debugging features. Additionally, the buttons above the viewer are used to package and download applications as well as to open the blocks editor (see Figure 2-4).	20
2-2	The completed user interface of a simple application for joining shared bulletin boards and posting messages for other users to see.	21
2-3	The configurable properties for a button component shown in the application designer.	22
2-4	An empty blocks editor. To build an application users select blocks from the panel on the left and drag them into the workspace.	23
2-5	The built-in block drawer for list handling operations. Blocks which return values are shown with plugs on their left while blocks that modify existing structures have the call decorator and have dips and bumps on their tops and bottoms so that they can be strung together as a series of operations.	24
2-6	The property setter for the text of a button. Property setter blocks have sockets on their right side, which can be filled with the plug of a property getter or value block.	25
2-7	An event handler for a button that pops up an input dialog when clicked.	26
2-8	An architecture diagram.	27

3-1	User interfaces for poll creation and voting.	31
3-2	Viewing vote totals on both the poll creation and voting screens. . . .	32
3-3	The method call block for sending a message to other players using the GameClient component.	35
3-4	Blocks to create new voting categories. When a user clicks on the new category button he or she is shown a text prompt. After inputting a category name, the program will make a server request to create a new public instance with the selected name.	39
3-5	Event handlers for retrieving the list of polls owned by the user. After selecting a category, the GameClient component sets its instance ID to the name of the category. When SetInstance completes, it triggers the InstanceIdChanged event handler. Server commands automatically include the current instance ID in their requests. Thus, invoking the Get My Polls server command will only return the polls for the selected category.	40
3-6	Event handlers for the buttons that are used to close, delete, and create polls. Each handler uses variables which are globally accessible to the program as the arguments to its associated server command. The server commands are performed asynchronously and trigger the ServerCommandCompleted event when they finish.	41
3-7	An example of using the selection from a ListPicker component to trigger a server command. The global variable receivedPolls has the same ordering as the list of strings that serve as the poll picker's elements. This decouples the information about a poll that is displayed to the user from information kept private by the program.	42
3-8	Error handling in the voting program. The server module was designed to handle multiple identical responses and to return human readable error messages. This means that simply informing users of errors is sufficient for them to handle the errors on their own.	43

3-9	The blocks used to retrieve the polls in the voting program. When the GetMessage function returns it parses its response into individual messages and triggers the GotMessage event handler for each one. In this example, new polls are added to a global list that is used to populate a ListPicker component.	44
3-10	The ServerCommandSuccess event handler and the blocks used to update the user interface with the response from a cast vote command. All server commands return App Inventor lists as their response.	45
5-1	Viewing the FreeFood bulletin board.	62
5-2	The Bulletin Board message reading loop. Bulletin Board uses a Clock component to make message requests every 10 seconds.	63
5-3	The event handler for new messages. When a new message is received, its sender and content are merged into a single text value and added to the bulletin board display.	64
5-4	The MoBulls and Cows game after submitting the correct sequence. Each guess is sent to the server when the player hits the “Submit Guess” button. The server then checks the guess against the correct sequence and returns the number of “bulls” and “cows”. If the player wins, the server will update his or her score statistics and send them back with their final score.	65
5-5	The blocks used to submit a new guess to the game server. The game first checks to make sure that the player has not previously tried the same guess and then submits it to server.	66
5-6	Viewing the scoreboard for MoBulls and Cows. High scores and statistics are kept track of by the server in a scoreboard that is stored with the game instance. Programs can request the scoreboard with a server command.	67

5-7	The Amazon program after looking up a book by keyword. The game server accesses the Amazon E-Commerce Services to perform a query for the keyword and returns any books it finds to the program.	68
5-8	The entire blocks workspace for the Amazon program. By returning book information in the same format for both keyword and ISBN searches, all server command responses can be handled with the same blocks.	69
5-9	The user interface after two additional players have joined a new game. The “Start Game” button is activated and will send a server command to begin the game if clicked.	72
5-10	Androids to Androids after submitting a card. Players choose cards from their hand using a ListPicker component and submit them to the leader. In response to the card submission, the server will replenish the player’s hand with a new card drawn randomly from the deck and send it back to them.	73
5-11	The Androids to Androids NewLeader event handler. The leader has a different set of actions than other players during a round. This event handler ensures that only necessary user interface elements are enabled for each player.	75
5-12	The GotMessage event handler in Androids to Androids. Depending on the type of the message received, the event handler calls another procedure with the message contents as the argument. These procedures know the specific format and ordering of the contents of their message type.	78
6-1	The user interface of StarLogoTNG. The blocks of StarLogoTNG closely resemble the blocks in App Inventor. A real time view of the running program is shown in the top right.	80

6-2	A procedure in Kodu. Kodu uses when and do blocks which operate equivalently to event handlers and method calls in App Inventor. Nested when blocks are used to implement conditionals.	81
6-3	The user interface for Alice. Game creators define animations and event handlers which control the interaction of characters in a 3D environment using the guided storyboard. The display of semi-colons and brackets is an optional feature of Alice that is used to transition users to Java or other written languages. The 3D animation displayed at the top of the window can be run to see the effect of changes to the storyboard during development.[4]	82

Code Listings

3.1	The voting module's server command to create a new poll. (Excerpt from A.13)	34
3.2	The command dictionary for four different custom modules. (Excerpt from A.9)	37
4.1	Adding dynamic properties to a Message object. (Excerpt from A.13)	48
5.1	Server code required to perform a search by keyword. (Excerpt from A.10)	70
5.2	The custom server command for submitting a noun card to the leader. (Excerpt from A.11)	73
A.1	server.py - The game server application file. Includes the request handlers for server requests.	89
A.2	server_commands.py - Contains the server command dictionary and built in server commands relating to instance management.	113
A.3	utils.py - Helper utility functions for input sanitizing and database operations.	118
A.4	game.py - The game database model.	121
A.5	game_instance.py - The game instance database model.	123
A.6	message.py - The message database model.	128
A.7	card_game.py - A library for handling card games in a game instance.	130
A.8	scoreboard.py - A library for keeping track of a per instance scoreboard.	140
A.9	commands.py - The command dictionary for custom modules.	145
A.10	amazon_commands.py - Amazon server commands.	146

A.11	ata_commands.py - Androids to Androids game commands.	148
A.12	bac_commands.py - Bulls and Cows game commands.	155
A.13	voting_commands.py - Voting commands.	158
B.1	GameClient.java - The GameClient component.	165
B.2	GameInstance.java - A container for information pertaining to game instances.	188
B.3	YailList.java - The AppInventor collection primitive.	191
B.4	JsonUtil.java - Utility functions for converting JSON to data repre- sentations understood by AppInventor.	195
B.5	WebServiceUtil.java - Utility functions for making POST commands from Android applications.	198

Chapter 1

Introduction

The increased prevalence and sophistication of mobile devices has created an exciting range of possibilities for mobile applications and games. In Europe and North America, most teenagers now own mobile phones and smart phones continue to represent the fastest-growing segment of the mobile device market[13]. The spread of this technology creates great opportunities to build mobile applications that enhance productivity, supplement learning, provide entertainment and connect people to information wherever they are. However, most of these users are left at the mercy of a relatively tiny population of sophisticated developers for the applications that they want. Additionally, mobile phone application development has continued to be excluded from Computer Science education due to the difficulty of working with historically closed and difficult to debug development environments.

Computer games have been used in classrooms to provide motivating examples and give students an exciting avenue for instruction[9]. Unfortunately, the complexity of building computer games often creates an insurmountable barrier for introductory students. Mobile games, on the other hand, do not suffer from this shortcoming because much of their entertainment value derives from interaction with others rather than advanced graphics or quick gameplay. Web enabled mobile phones provide the capability to make programs and games with real time interaction that can replace the complex game characteristics seen on platforms

such as video game consoles and desktop computers[2]. Enabling users to interact with each other using their phones provides even more room for application ideas to grow.

Typically, multi user games consist of a single server and a large number of clients[12]. Unfortunately, the complexity of client/server interaction with mobile devices makes creating multi user applications very difficult. To address these issues and empower mobile phone users to create social applications and games, I created a multiplayer framework for App Inventor. The framework includes an App Inventor component with a suite of client side operations that communicate with a game server in order to create and play games. With these operations, users of App Inventor can create mobile applications to communicate and coordinate with each other, access external web services, perform computation in the cloud and maintain game and user state online.

1.1 App Inventor for Android

The client side of the framework is implemented specifically for the App Inventor for Android system, a project currently underway at Google Research[1] that aims to turn mobile phone users into mobile application creators. App Inventor builds on previous work done on graphical programming languages such as StarLogoTNG[10] and the Openblocks library[11] to provide an application development framework that give users without coding experience the ability to create mobile applications.

In the fall 2009, App Inventor was used in a pilot program at a dozen universities[1] as a tool to help teach students about a range of topics related to computer science, digital privacy and the importance of technology in society. During the semester, students created a variety of simple phone applications and explored some of the difficulties of developing on a mobile platform. Following this pilot program, Google Research has continued to work on App Inventor as more classes have started to use it in the spring.

App Inventor works by packaging the complexity of user interface widgets and phone hardware features into easy-to-use components. The functionality of these components is exposed to application developers via graphical building blocks, instead of with written code. Just like putting together a puzzle, users of App Inventor can snap together these blocks to create mobile phone applications without the need to write code or understand the complexities of deploying applications.

1.2 Games in App Inventor

Existing App Inventor components focus on local application behavior such as the appearance of on screen components and direct user input. Programs generally consist of a single screen and have little ability to interact with other programs or access functionality outside of the device. The multiplayer framework widens this focus by creating a GameClient component and a game server implemented using Google App Engine. The use of App Engine as a server platform allows application developers to easily customize and deploy their own servers. A default game server is hosted by Google for application creators to use as a testing and development environment. However, given that the process of starting and customizing one's own server is reduced to the execution of a Python program when using App Engine, it is expected that most application developers will deploy their own servers.

The GameClient component follows the pattern of existing App Inventor components by packaging the code required to complete game server requests into a set of method calls and event handlers that are exposed as blocks. When the server call blocks are used, the component sends requests using the phone's mobile data connection¹. Additionally, data that dictates the flow of a game and its list of players is automatically processed by the client to provide the application with helpful events such as when a player enters or leaves a game and when an invite to join

¹The marshalling and unmarshalling of data into formats that can be understood by both App Inventor and the web server are automatically handled and invisible to the user.

another game has been received.

Without modification, the game framework provides a default implementation that includes basic game management, message handling, and access to built-in extensions such as a scoreboard and a card game manager. The BulletinBoard application shown in Section 5.1 demonstrates an application created using the unmodified server deployment. However, the ability to add custom behavior to the game server is a very important design goal for the game framework. Consequently, the framework was built to provide extensibility through custom modules that can be added to a game server installation without making any modification to the baseline code.

During the pilot program, a web database component called TinyWebDB was provided to students along with its server source code. During the semester, a number of students modified this server code to overload the database's simple get and put commands. Eni Mustafaraj, a professor at Wellesley, even worked with the Google team to create a custom component that allows applications to access a server that implemented a voting application. This process involved many hours of effort and hundreds of lines of code to both create a custom component to serve as a client and to implement the needed request handlers and database models on the server side. This is not only a laborious approach, it is impossible for most users because creating a new component in App Inventor requires modifications to production Google services. Chapter 3 will walk through a reimplementing of the polling application built at Wellesley. The new version requires only a single Python file with less than 100 lines of code be added to the game server. On the client side, no new components or other modifications are needed.

1.3 Thesis Summary

The next chapter provides an introduction to the App Inventor system. The following chapter is a walkthrough of the creation of a polling application. Following the walkthrough is a system overview that gives an in-depth look at the GameClient

component and the server API.

After the system overview are descriptions of four other example application that I have built to demonstrate the various use patterns of the framework. Then, Chapter 6 presents the impact on other work in the areas of graphical programming and mobile game creation. The final chapters discuss possible future extensions to the framework and list the contributions of this thesis.

Chapter 2

Introduction to App Inventor

This chapter gives an introduction to the two-part application creation process in App Inventor. First, application creators design their user interface with the application designer, which is run in a web browser and hosted by Google's App Inventor server. Then, creators must define the program logic in the App Inventor blocks editor. To use the multiplayer framework, users simply include the Game-Client component in their App Inventor project and use the component's blocks to make calls to the game server running on App Engine.

The App Inventor server, which runs the application designer, also contains the component code and compiles user projects. These pieces all work together to enable users to easily design, build and package applications that can run on any Android phone.

2.1 The Application Designer

The first step in creating an App Inventor project is to add components and lay out the user interface using the application designer. Figure 2-1 shows a blank application designer window running in a web browser. On the left is the component palette showing a collection of basic user interface components such as buttons, labels and text input boxes. Users add components to their projects by dragging them from the palette on the left to the viewer in the middle. Figure 2-2 shows a

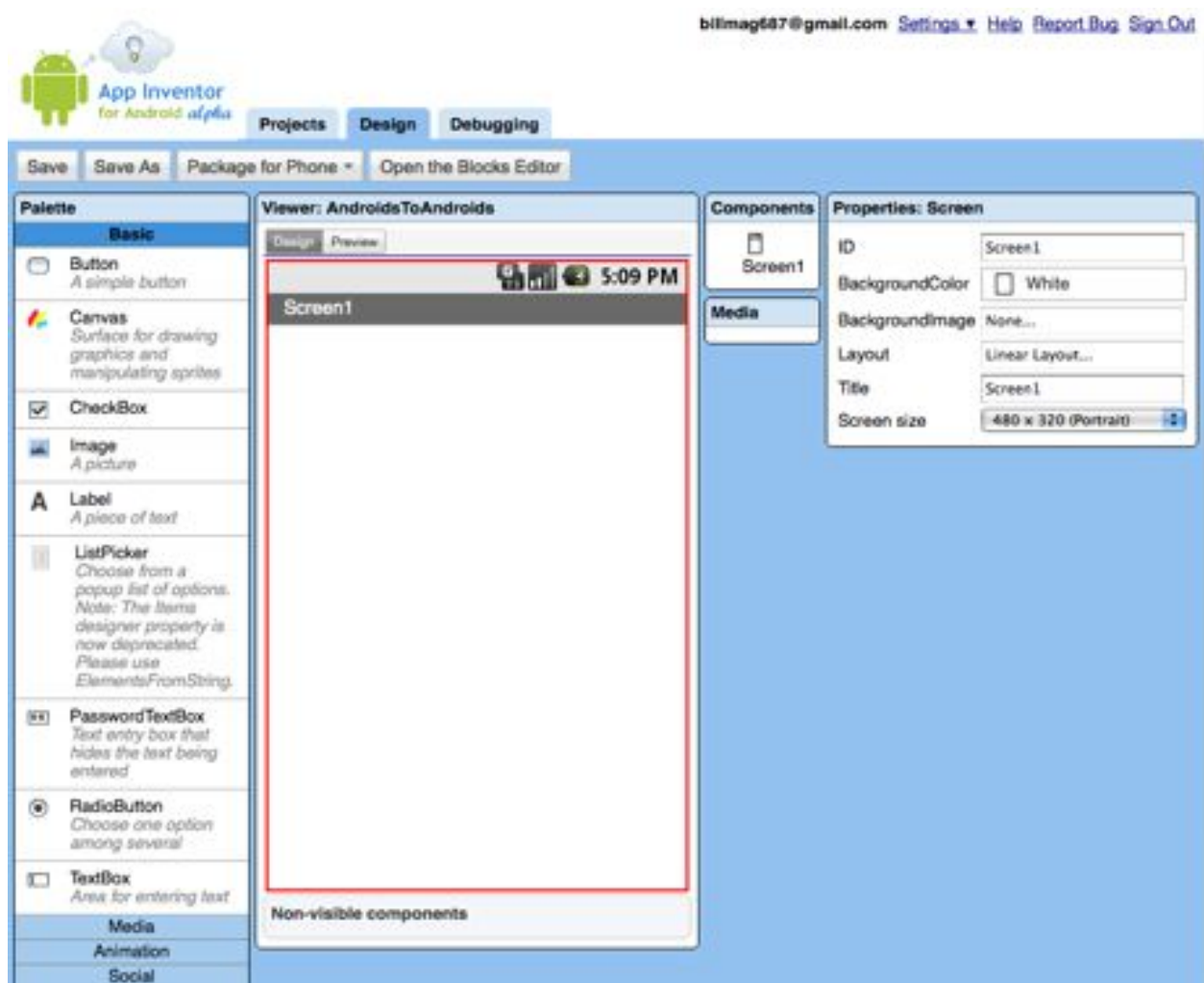


Figure 2-1: An empty application designer window. The application designer is just one part of the App Inventor web interface. The tabs visible across the top provide access to project management, sharing and debugging features. Additionally, the buttons above the viewer are used to package and download applications as well as to open the blocks editor (see Figure 2-4).

completed user interface design for a web enabled bulletin board program.

The application designer allows users to define both the layout and the initial attributes of their components. Whether a property is allowed to be changed is determined by the creator of the component. Component configuration can include both aesthetic and behavioral properties. As an example, the properties panel for a button is shown in Figure 2-3.



Figure 2-2: The completed user interface of a simple application for joining shared bulletin boards and posting messages for other users to see.

2.1.1 Non-Visible Components

In addition to the user interface widgets, the designer also includes non-visible components. Each non-visible components falls into one of the following categories:

- Sensors - Include all components that access phone hardware features such as the GPS or accelerometers.
- Notifiers - Are capable of popping up alerts or writing to the phone's activity log. Notifier components are generally not visible at startup, but can contribute to the user interface through the form of text input dialogs or other popups.



Figure 2-3: The configurable properties for a button component shown in the application designer.

- Clocks - Provide access to time related functions and a timer that can be set to periodically trigger events.
- Activity Starters - Allow a program to start or use other installed applications on the phone. Some examples include the barcode scanner and text to speech components.
- Web Services - Include the GameClient component, a web database with simple put and get operations, a Twitter component and the original Voting component.

The GameClient component includes blocks for completing requests to the game server. When a call is made using GameClient, a new connection is opened to the game server and a POST request is made. The game server reply is interpreted by the GameClient and used to update its own properties or trigger events related to the game.

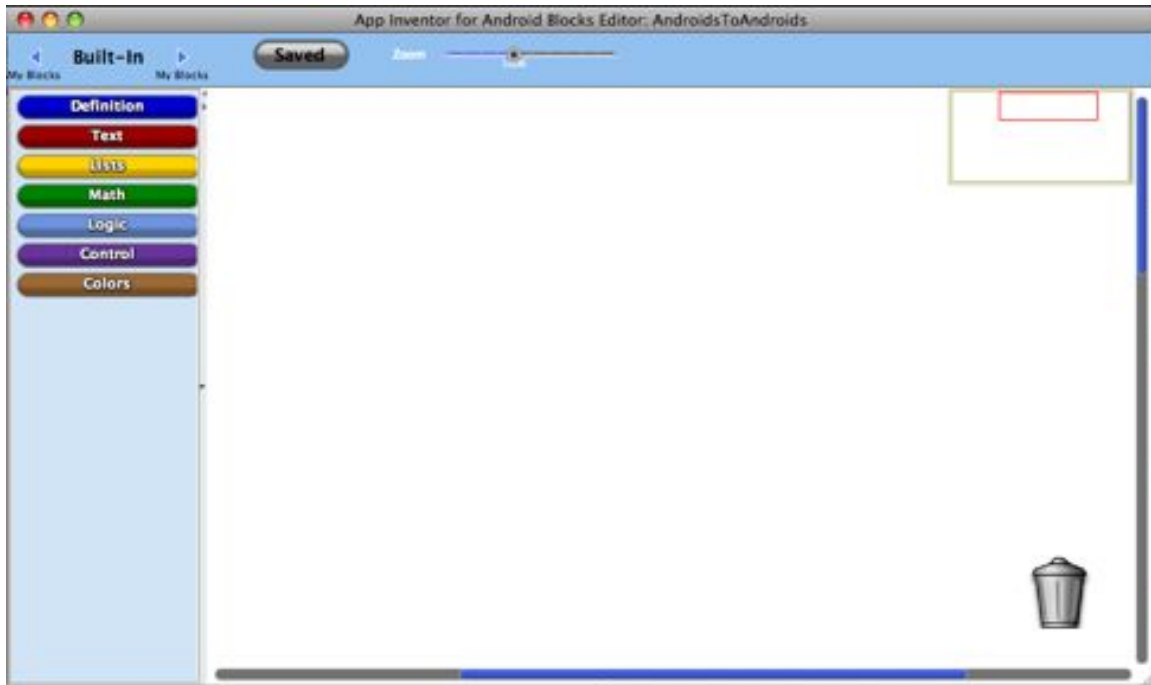


Figure 2-4: An empty blocks editor. To build an application users select blocks from the panel on the left and drag them into the workspace.

2.2 The Blocks Editor

After all the components for a project are selected, users must create the programming logic for their programs in the blocks editor. An empty blocks workspace in the blocks editor is shown in Figure 2-4.

The blocks editor combines related code blocks into drawers. Users can drag blocks from these drawers into the workspace in order to add them to their projects. Built in drawers such as the one seen in Figure 2-5 provide the programming primitives that are used to create applications. These primitives are split up into categories based on their purpose. For example, text blocks are used to operate on strings while math blocks provide functions that operate on numbers.

An important abstraction capability in App Inventor is the ability to define procedures and variables which can be called from procedures and event handlers. The blocks to create these are in the definitions drawer.

Finally, each component has a drawer with all of the blocks defined for it by the component creator. The blocks in a component drawer come in four flavors: event



Figure 2-5: The built-in block drawer for list handling operations. Blocks which return values are shown with plugs on their left while blocks that modify existing structures have the call decorator and have dips and bumps on their tops and bottoms so that they can be strung together as a series of operations.

handlers, method call blocks, property getters and property setters.

2.2.1 Property Getters

Property getter blocks have a plug on their left side and return the value of various readable properties of components. Generally, these values are simple field getters, but in some cases they actually hide complex operations. The GPS getters on the Location Sensor component are a good example of this. To the user in App Inventor, the process of accessing the GPS is just as easy as reading the text of an input box. Abstracting this process ensures that users are not bogged down by the complexity of accessing information.

2.2.2 Property Setters



Figure 2-6: The property setter for the text of a button. Property setter blocks have sockets on their right side, which can be filled with the plug of a property getter or value block.

Property setters change the properties of a component to the value represented by the blocks that are plugged in the sockets on their right side. These sockets are shaped like the plugs on property getter blocks to indicate that they can be snapped together.

2.2.3 Event Handlers

Events are triggered whenever an event such as the click of a button, the return of a server call or a change in the accelerometer values occurs during the course of operation of a program. The body of an event handler block defines the actions to be taken when the event occurs. Figure 2-7 shows an event handler for a button that pops up a text input dialog when the button is clicked.

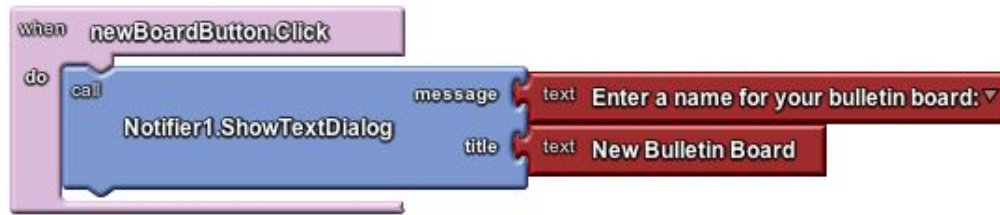


Figure 2-7: An event handler for a button that pops up an input dialog when clicked.

2.2.4 Method Calls

Method call blocks can be used in both user defined procedures and event handler bodies to define appropriate responses to events. These blocks encapsulate component operations and generally trigger further events in the process. The ShowTextDialog block in Figure 2-7 is an example of a method call that receives two arguments. The arguments are passed to the component to define the appearance of the text input box that is created.

Method call blocks have dips and bumps on their tops and bottoms to indicate that they can be stacked on top of each other to perform multiple actions in response to an event. These blocks do not return values as programmers might expect a method to. This is made apparent visually by the lack of a plug on the left side of the method call block.

2.3 Framework Overview

The multiplayer framework consists of a non-visible GameClient component, a game server running on App Engine and utility classes in App Inventor which perform web service calls and convert data between App Inventor types and a format that is understandable to a server. Every application that uses the game server must include a GameClient component and use the method call blocks to make server requests. The basic architecture of this system is shown in Figure 3-3.

The GameClient includes built in method call blocks which make requests to perform actions such as joining game instances, inviting new members, sending

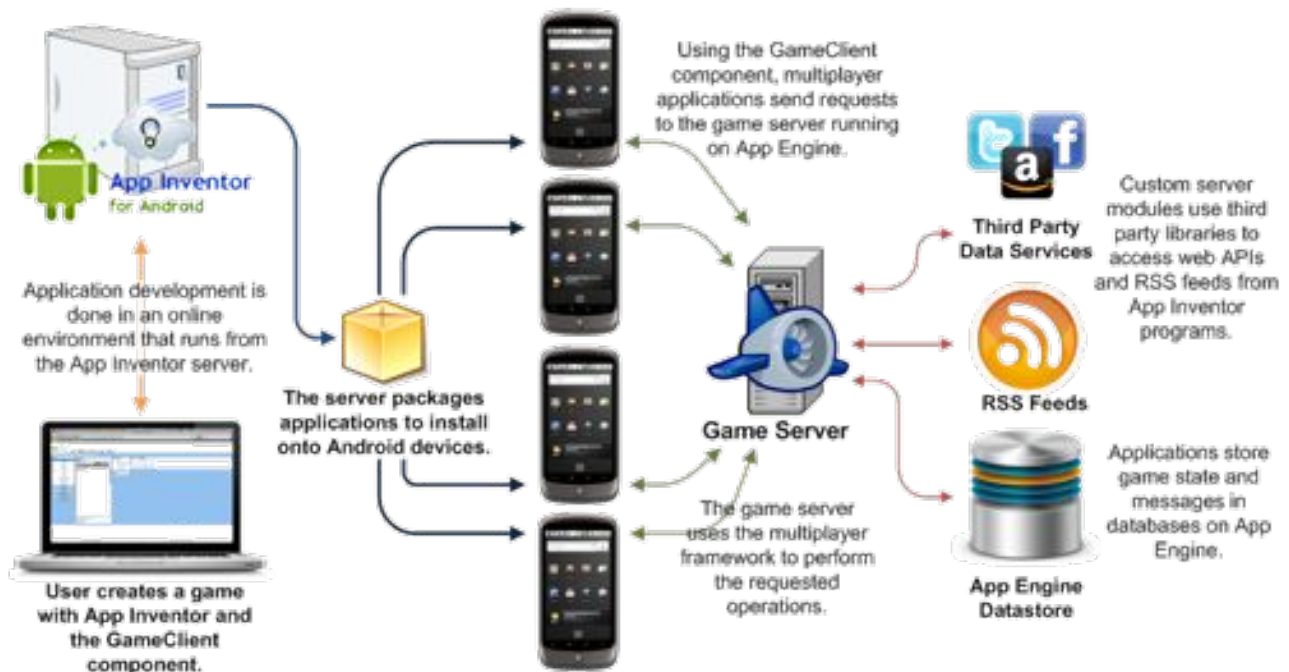


Figure 2-8: An architecture diagram.

messages to other players and executing custom server commands. The game server defines a request handler for each of these actions which accept POST requests from the GameClient and reply with JSON objects containing the result of executing the server command.

To store data, the game server uses the App Engine data store. Information about games is organized into three levels:

- **Game** - Each application has one Game object in the data store which is identified by the game ID property of its GameClient component. The game ID operates as a namespace, allowing each application to maintain its own set of GameInstances. Game objects are also used to perform queries related to GameInstance objects such as finding the list of instances that a player has joined or been invited to.
- **GameInstance** - A GameInstance represents a group of players participating in a single game. A Game object can have any number of GameInstance children, but each GameInstance must have a unique instance ID among its

siblings. Together, the game ID and instance ID uniquely describe a `GameInstance` object.

- **Message** - Message objects are used to send information from one player to another in a `GameInstance`. Messages can be created in App Inventor by using the `SendMessage` method call block or created by custom server commands.

The game server can also be augmented with custom modules written by application creators that define server commands to be called with the `GameClient`'s `ServerCommand` block. These custom modules have full access to the datastore and can use third party libraries and data sources to give programs access to outside information or execute complicated game logic.

2.4 Packaging and Running the Application

The final step is to package and run the application. The user kicks off this process by clicking on the **Package** button in the application designer (Figure 2-1). The App Inventor server then orchestrates the saving, compiling and building of the application. Ultimately, the server delivers a fully functional application that can be installed on any Android phone and shared with friends.

Chapter 3

Building a Game

This chapter explains the process of building a multi user polling application called Poll Your Friends using the multiplayer framework. It will start off with an introduction to the use of the program, move to a discussion of the custom server module and finally show the blocks that make up the client. A polling application is uniquely suited to the multiplayer framework because it uses interaction with others to derive enjoyment and utility. Poll Your Friends consists of three parts:

- Poll Your Friends - Displays polls to users and allows them to submit votes. Once a user has voted in a poll or the poll is closed by its creator, the vote counts for each option can be viewed.
- Poll Your Friends Creator - Allows users to create new polls for others to vote on. Poll creators can also view the vote counts in their polls, close them to new votes or delete them entirely.
- Voting custom server module - The server module is a python file running on the server, which defines seven different server commands. These commands can be invoked by the GameClient component by using a method call block.

3.1 User Interfaces

In order to simplify the user interface, Poll Your Friends splits poll management and voting into two separate Android programs¹ The user interface for each program is shown in Figure 3-1. Together, the two programs provide a user with the ability to do the following:

- Create new categories for polls.
- Create new polls by defining a category, a question and between two to five response options.
- Close and delete his or her own polls.
- View real time results of his or her own polls.
- View polls created by other users.
- Vote in open polls.
- See the results of polls he or she has voted in.
- View the results of closed polls.

Once a poll has been created, it can be viewed and voted on by others. Users are uniquely identified by the email account that is registered with their Android phone and disallowed from voting more than once in the same poll. Similarly, poll ownership is tracked by recording the email address of the poll creator.

After a user has voted in a poll, he or she is allowed to view its current vote totals but not change his or her vote. When the poll is closed by its creator, all users are allowed to see the final vote counts until the poll is deleted. Figure 3-2 shows the screens for viewing vote totals in both the voting program and the poll creator program. The owner of a poll can close or delete it by pressing the buttons below the options list.

¹App Inventor currently limits projects to a single form or user interface screen. This means that a two form application must be installed as two separate programs. This is likely to change as App Inventor matures as a platform.



(a) The PollYourFriendsCreator program. This screenshot shows the process of creating a new poll. The user has selected a poll category and defined his or her question. After submitting the poll to the server, it will be available for others to vote on.



(b) The PollYourFriends program. The user has selected the newly created poll and is ready to vote on it. The “Submit Vote” button will become enabled once the user has selected an option.

Figure 3-1: User interfaces for poll creation and voting.

Polls are split into different categories depending on their subject. All of the polls hosted on the demonstration server are made public, however, access control can be added with the invite system (see ??). Players select from available lists and categories by using a component called a ListPicker. List Pickers appear on the user interface as buttons. When pressed, they display a list of strings and allow the user to select one.

The screenshots also show a number of disabled buttons. A button’s state changes during program operation to define allowable actions. An example is seen in Figure 3-2 (b), where the “Submit Vote” button is disabled before a user



(a) Poll creators can view the vote totals for their polls at any time and are allowed to close and delete their polls.

(b) When a user selects a poll from the list of open polls the server will check to see if he or she has voted in that poll.

Figure 3-2: Viewing vote totals on both the poll creation and voting screens.

has selected an option. This keeps the user from accidentally submitting an empty vote to the server. (As a backup, the server module has been written to deal with incorrect user input, but it improves usability to direct the user's behavior in the client as well.)

3.2 The Voting Server Module

The voting server module is a Python file written for the game server that defines seven server commands in approximately 75 lines of code. The code for this module is shown in Code Listing A.13. Two of the commands are used by the voting program and the remaining five are used by the poll creation program. The com-

mands are as follows:

- **Get Results** - Used by the voting program to request information about a poll's status and see if the user has already voted in it. While the client is waiting for the request to complete, the "Submit Vote" button is disabled. If Get Results responds without the poll results, the client knows that the user is allowed to vote and re-enables the button.
- **Cast Vote** - Submits a vote from the voting program. This command confirms that the requested poll is still open and that the player has not yet submitted a vote before recording it into the database. It then returns the current poll results to the client.
- **Get Poll Info** - Returns detailed information about a particular poll. Unlike Get Results, this can only access polls that were created by the requesting player and always returns the current vote totals.
- **Get My Polls** - Returns a listing of all polls created by a player in a particular category. This listing is used to populate the "View My Polls" ListPicker in the poll creator program. When a user selects a poll from this list a Get Poll Info command is sent to retrieve more information.
- **New Poll** - Creates a new public poll for others to vote on.
- **Close Poll** - Closes polls to new votes and allow all users to see the final vote counts.
- **Delete Poll** - Purges a poll from the server and removes its vote history.

Every server command must accept three parameters: a database model, the email address of the requesting player, and a list of arguments. Generally, server commands expect to receive a game instance as the database model. The game framework uses instances to represent a particular subset of a game's players and to serve as a parent for messages passed to members of that instance.

The voting module uses game instances to separate its polls into categories. As mentioned before, these instances are made public and allow an unlimited number of players to join them. However, game instances can also be made private and limit their membership to a particular number of people or only those who have been invited.

The code for the Make New Poll command is shown below. It first validates the inputs to make sure the question is not empty and an acceptable number of options has been provided. The procedure then initializes a Message object with an empty recipient and stores it in the database.²

3.1: The voting module's server command to create a new poll. (Excerpt from A.13)

```
1 def make_new_poll_command(instance, player, arguments):
2     """ Make a new poll.
3
4     Args:
5         instance: The game instance to add the poll to.
6         player: The email of the player creating the poll.
7         arguments: A two item list containing the question and a
8             second list of 2-5 options.
9
10    Returns:
11        Returns a list with information about the poll just created.
12        See get_poll_return_list for its format.
13
14    Raises:
15        ValueError if the player is not in the instance.
16    """
17    instance.check_player(player)
18    if not arguments[0]:
19        raise ValueError('Question cannot be empty')
20    size = len(arguments[1])
21    if size < 2 or size > 5:
22        raise ValueError('Incorrect number of options for poll. ' +
23                          'Must be between two and five.')
24
25    poll = Message(parent = instance, sender = player,
26                  msg_type = 'poll', recipient = '')
27    poll.put()
28    arguments.append(poll.key().id())
29    poll.content = simplejson.dumps(arguments)
30    poll.votes = [0] * size
31    poll.open = True
32    poll.voters = ['']
```

²The empty recipient field means that the message can be fetched by any user that has joined the game instance by selecting the category.

```
33 poll.put()
34 return get_poll_return_list(poll)
```

3.2.1 The Message Model

Sending messages is the main form of communication among the players in a game instance. Each message is created with a type string, a list of recipients, and a list of contents. New messages can be created directly in server modules or sent from applications with the SendMessage block (see Figure 3-3). Message contents are stored on the server as JSON. When a message is sent back to a client, the JSON is parsed and converted into App Inventor lists. The contents can then be accessed with the list operation blocks shown in Figure 2-5.



Figure 3-3: The method call block for sending a message to other players using the GameClient component.

In the server voting module, polls are represented by messages in the database. The contents field of each poll is a three-item list containing the poll question as the first element, a list of the options as the second element and a numerical identifier for the poll as the third element. The numerical identifier is used by the client to identify polls to the server, but never exposed to the application user. The voting program finds open polls by using a GetMessages call block. Each received poll triggers a GotMessage event with its message type and contents as arguments. These polls are then added to the poll ListPickers for the user to select.

In addition to its default fields, each message can also store dynamic properties that can be assigned to it at runtime. The votes, open, and voters fields on lines

30-32 of Code Listing 3.1 are examples of dynamic properties for a poll. Dynamic properties are stored in the database along with the static properties and can be accessed and modified by other server commands. This flexibility allows for the existing database models to be used for a wide variety of server modules on the same server without modifying the base server code.

3.2.2 Input Validation and Error Handling

Server commands are implemented using transactions. At the beginning of their execution, most server commands perform input validation and permission checking before continuing. In the Make New Poll example, both the question and the options are checked before making any costly database operations. If the arguments provided to the server command are invalid, a `ValueError` is raised. Raising an unchecked error during a server command results in all actions performed during the request reverting their changes. The failed request will be reported back to the client and trigger a the `GameClient` component's `WebServiceError` event (shown later in Figure 3-8).

This transactional design makes server command logic simple because application creators can assume an all-or-nothing paradigm for the completion of their server-side actions. Despite this, a network failure could case a server command could complete successfully but not return its result to the client. For this reason server commands should be created in a way that allows them to be called multiple times without irreversible side effects. In the voting module, the only side effect of a server side action completing without the user being informed of its success is that he or she will not find out about the change until the next refresh of their data. However, since the server verifies that all requests are valid before allowing actions to continue, even if a program is acting with incorrect data it will not harm the stored data or affect other users.

3.2.3 Registering Commands

Before a custom server module can be accessed with the GameClient component, the commands must be registered with a request handler. This is done automatically when the application is started by reading in a command dictionary. Shown below is the command dictionary for the demonstration server running at <http://appinvgameserver.appspot.com>. It enables the server commands for four different applications, which are presented throughout this thesis.

3.2: The command dictionary for four different custom modules. (Excerpt from A.9)

```
1 custom_command_dict = {
2     # Androids to Androids
3     'ata_new_game' : ata_commands.new_game_command,
4     'ata_submit_card' : ata_commands.submit_card_command,
5     'ata_end_turn' : ata_commands.end_turn_command,
6
7     # Bulls and Cows
8     'bac_new_game' : bac_commands.new_game_command,
9     'bac_guess' : bac_commands.guess_command,
10
11     # Amazon
12     'amz_keyword_search' : amazon_commands.keyword_search_command,
13     'amz_isbn_search' : amazon_commands.isbn_search_command,
14
15     # Voting
16     'vot_cast_vote' : voting_commands.cast_vote_command,
17     'vot_get_results' : voting_commands.get_results_command,
18     'vot_new_poll' : voting_commands.make_new_poll_command,
19     'vot_close_poll' : voting_commands.close_poll_command,
20     'vot_delete_poll' : voting_commands.delete_poll_command,
21     'vot_get_poll_info' : voting_commands.get_poll_information_command
22     ,
23     'vot_get_my_polls' : voting_commands.get_my_polls_command
24 }
```

This command dictionary enables four different custom modules to operate on the server at the same time. No other code changes are required in order for a custom module to be successfully called from App Inventor.

3.3 Block Logic

After a project's components are selected, code blocks are used to define the behavior of the application. The voting application requires dozens of blocks to control its user interface and properly display polls, but the number required for server communication is relatively low. This section will first look at the block logic used in the poll creation program and then move on to the program used to perform voting.

3.3.1 Poll Your Friends Creator Blocks

The first action that a user takes after opening the poll creator program is to create or join a poll category. If others have previously created categories, users can join them and add new polls. However, if no categories exist or if users are not satisfied with any of the available choices, they must create a new category.

The category creation process is implemented with the two event handlers shown in Figure 3-4. When users click on the “New Category” button, they are greeted with a text input dialog box that prompts them to enter a new category name. Once they have entered a category name, the `AfterTextInput` event on the text dialog fires with the category name as its argument. If the user has inputted a non-empty string into the dialog box, a server call is made to make a new game instance with the selected name.

Retrieving and Displaying Polls

After a category has been selected, its polls are retrieved from the server. A category is activated by setting the `GameClient`'s instance ID to the name of the category. When the `GameClient` component makes a request, it includes the game ID³ and instance ID as arguments. Together, the game id and instance ID form a unique key that the server uses to retrieve the game instance from the database and pass it to the voting module's server commands.

³The game ID is set by the application creator and hard-coded into each application.



Figure 3-4: Blocks to create new voting categories. When a user clicks on the new category button he or she is shown a text prompt. After inputting a category name, the program will make a server request to create a new public instance with the selected name.

Unlike most other components, GameClient properties can only be modified with call blocks (as opposed to property setters). This emphasizes to program creators that setting a property in a game requires a server request. For example, when the SetInstance function is called, the component makes a request to the server to join the instance. If the server request succeeds, the InstanceIdChanged event triggers with the new instance ID as a parameter.

Figure 3-5 shows the InstanceIdChanged event handler for the poll creator. Once the instance ID has been set, the Get My Polls server command is automatically called.

Managing Polls

The three main actions in the poll creator are triggered with the Delete, Close and Submit buttons arranged horizontally below the poll options. The event handlers for these buttons are shown in Figure 3-6. Each of them invokes a ServerCommand to perform the requested action.

The Close and Delete server commands accept the number of the targeted poll

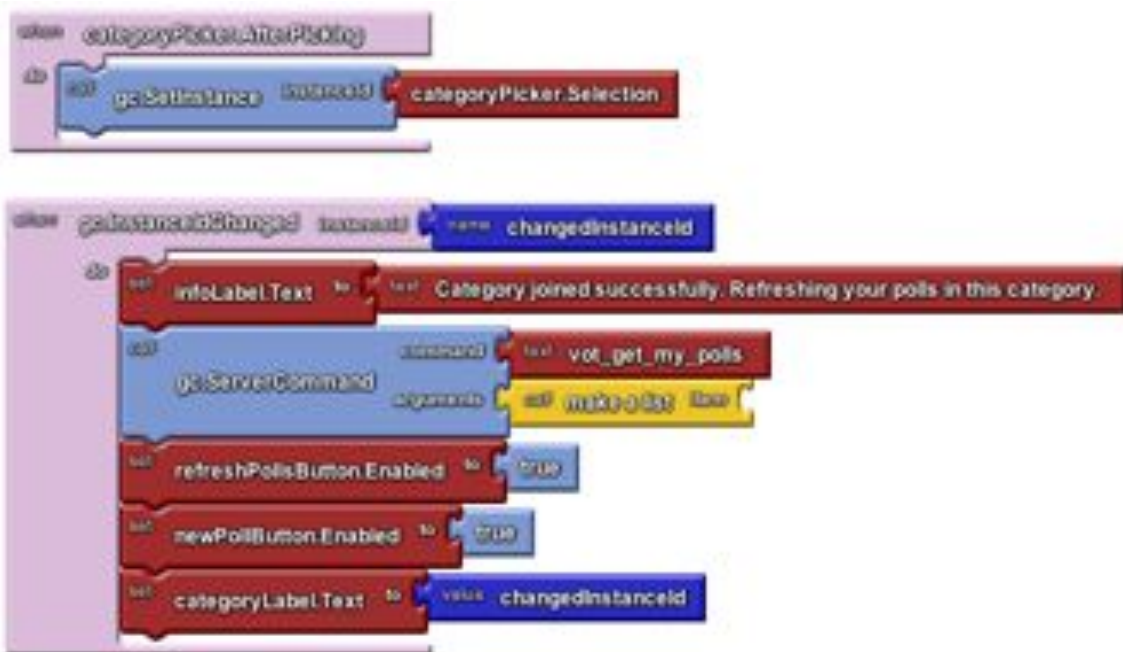


Figure 3-5: Event handlers for retrieving the list of polls owned by the user. After selecting a category, the GameClient component sets its instance ID to the name of the category. When SetInstance completes, it triggers the InstanceIdChanged event handler. Server commands automatically include the current instance ID in their requests. Thus, invoking the Get My Polls server command will only return the polls for the selected category.

as their only argument. Poll numbers are sent back from the server along with their associated question when a player requests their poll list. The questions and poll numbers are then stored in a global variable which can be accessed by other procedures.

When a user selects a poll question using a ListPicker, the index of the chosen question in the ListPicker's elements is used to retrieve the poll's ID number from the global received polls list. This approach is used to create a map from poll question to ID number⁴. Figure 3-7 shows the blocks for selecting a poll in the poll creation program.

⁴This is done through the use of nested list commands. Once dictionary support is written for App Inventor this particular design method will become much easier and more efficient.

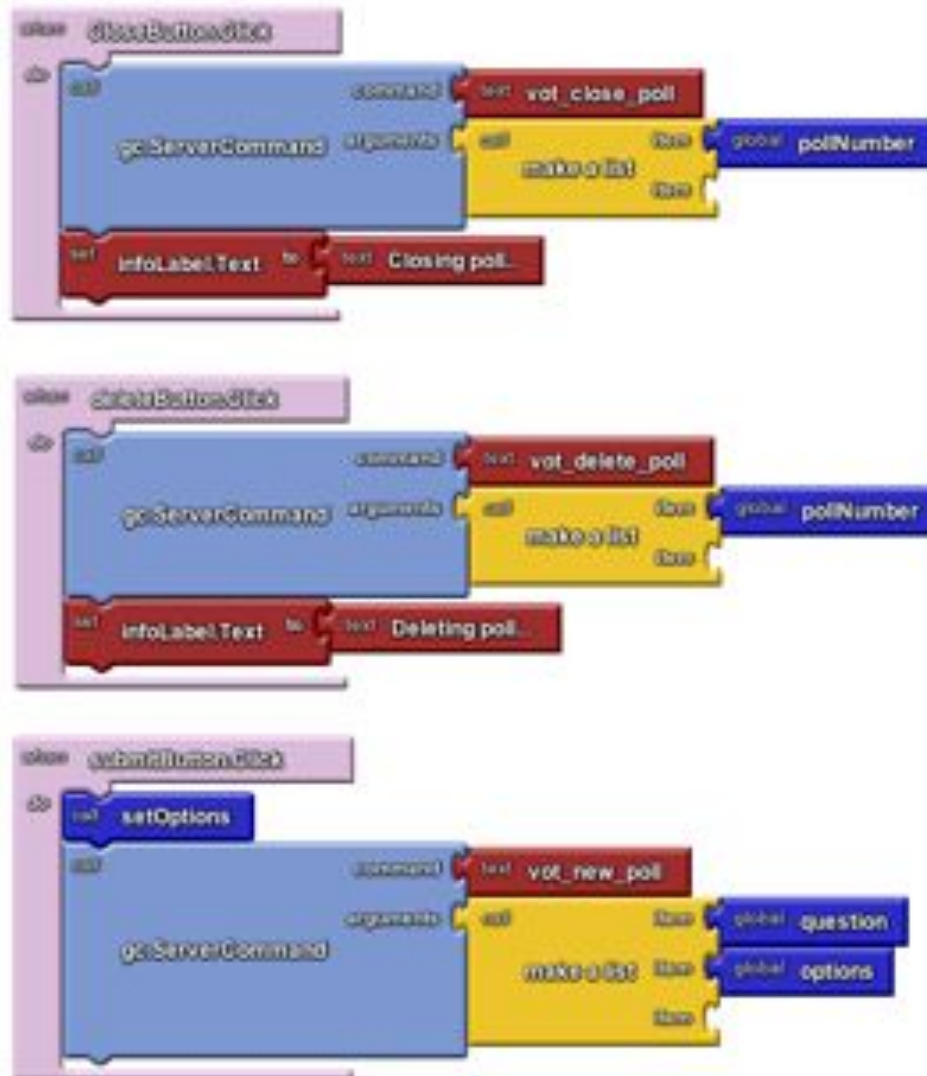


Figure 3-6: Event handlers for the buttons that are used to close, delete, and create polls. Each handler uses variables which are globally accessible to the program as the arguments to its associated server command. The server commands are performed asynchronously and trigger the ServerCommandCompleted event when they finish.

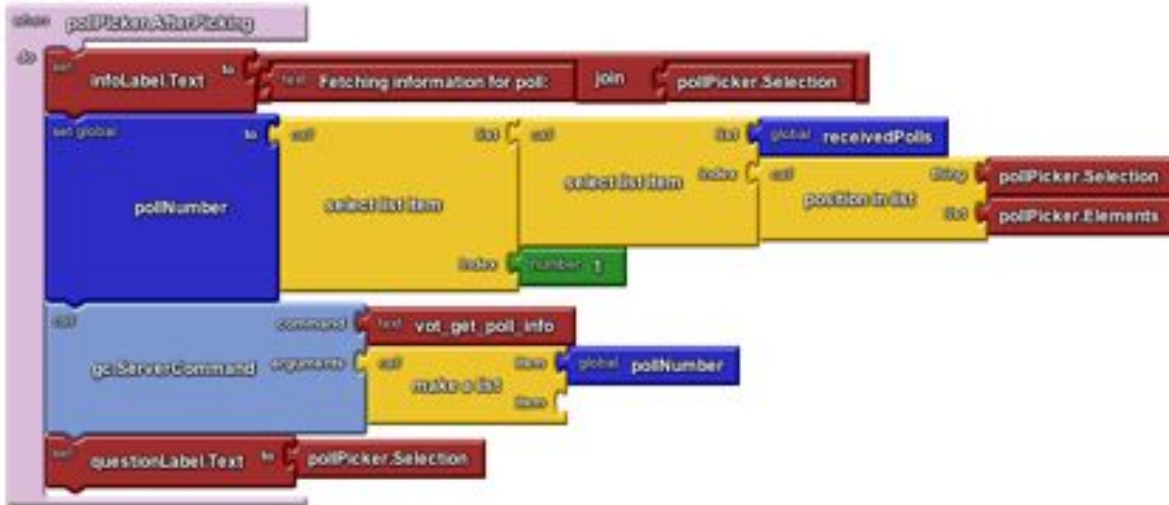


Figure 3-7: An example of using the selection from a ListPicker component to trigger a server command. The global variable `receivedPolls` has the same ordering as the list of strings that serve as the poll picker's elements. This decouples the information about a poll that is displayed to the user from information kept private by the program.

Web Service Errors

Given the inconsistency of mobile data connections, it is important that programs deal with connection issues and other problems with calls to web services. To help program creators, the `GameClient` component triggers `WebServiceError` events whenever the server aborts a transaction or when a connection failure occurs. Server error messages have all been designed to be human readable and provide the user with enough information to determine what has gone wrong. To make dealing with these errors less of a burden, the voting server module was designed to be able to handle repeated inputs of the same command from an out-of-sync client. Thus, as can be seen in Figure 3-8, the only response to a `WebServiceError` is to inform users that it has occurred. They can then retry their previous action after remedying any problem that arises.



Figure 3-8: Error handling in the voting program. The server module was designed to handle multiple identical responses and to return human readable error messages. This means that simply informing users of errors is sufficient for them to handle the errors on their own.

3.3.2 Retrieving Polls and Casting Votes in Poll Your Friends

Poll Your Friends is responsible for retrieving polls, casting votes and displaying results. Polls are retrieved automatically when players select a new poll category or when they manually click on the “Refresh Polls” button. The event handler for the refresh polls button is shown in Figure 3-9. Polls are retrieved using the GameClient component’s built in GetMessages call. The message type specified in Figure 3-9 is the empty string. This tells the server to not filter based on message type and instead return all messages that have been sent to the requesting player. The two message types for polls are poll and “closed_poll”. All polls are originally created with the message type poll, but are changed to “closed_poll” when the creator of the poll decides to disallow further voting. Because they are handled in different ways, the GotMessage handler checks the message type before processing the poll response.

After the user selects a poll to view, the client requests more information from the server about the status of the poll. If the user has not yet voted, the poll options will be presented and the user will be allowed to submit a vote. Once his or her vote is made, the server sends the current vote totals for each option back to the client application. The ServerCommandSuccess event handler and the procedure to update the user interface with vote counts is shown in Figure 3-10. The

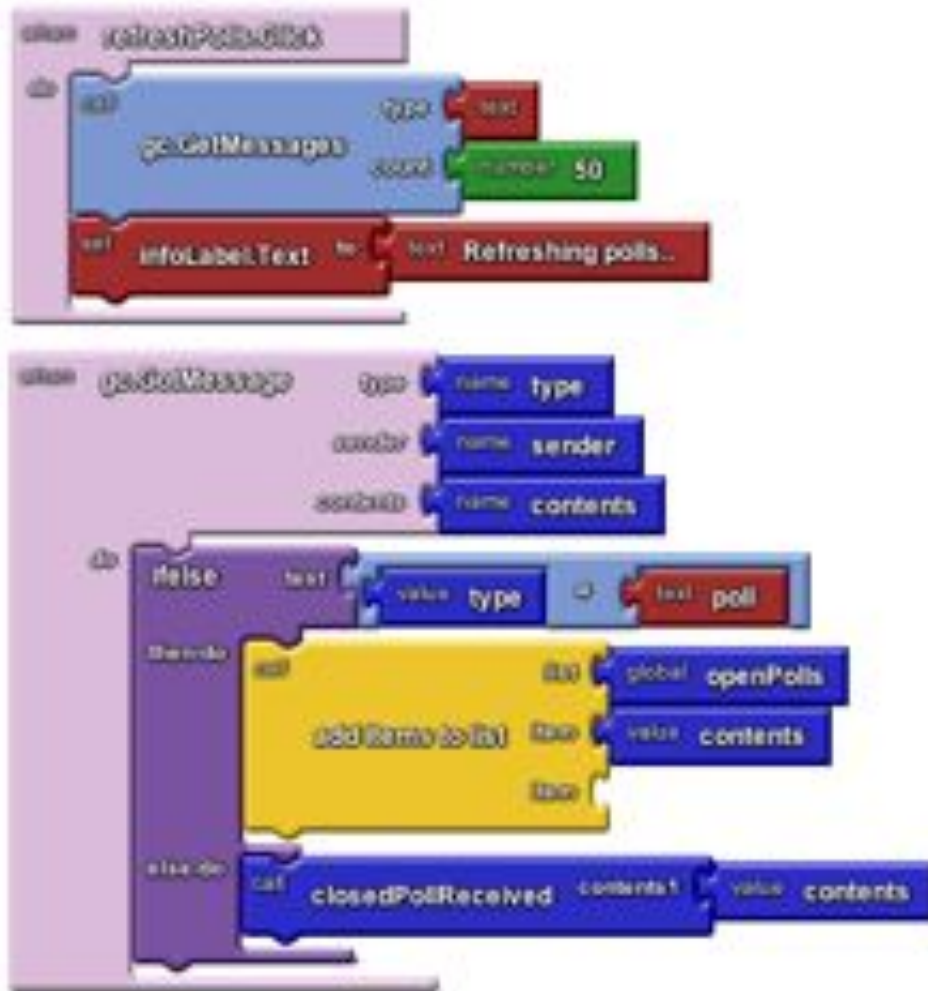


Figure 3-9: The blocks used to retrieve the polls in the voting program. When the GetMessages function returns it parses its response into individual messages and triggers the GotMessage event handler for each one. In this example, new polls are added to a global list that is used to populate a ListPicker component.

event handler checks the type of command that the response is for and invokes the appropriate procedure. If the response was for a Cast Vote command, the client knows that the response contains a message from the server as its first element and a list of the vote counts as the second element. The procedure then selects these items from the response list and updates the user interface.

Now that the application creation process has been explained, the next chapter will provide an in-depth look at the details of the server API and the blocks that the GameClient component exposes.

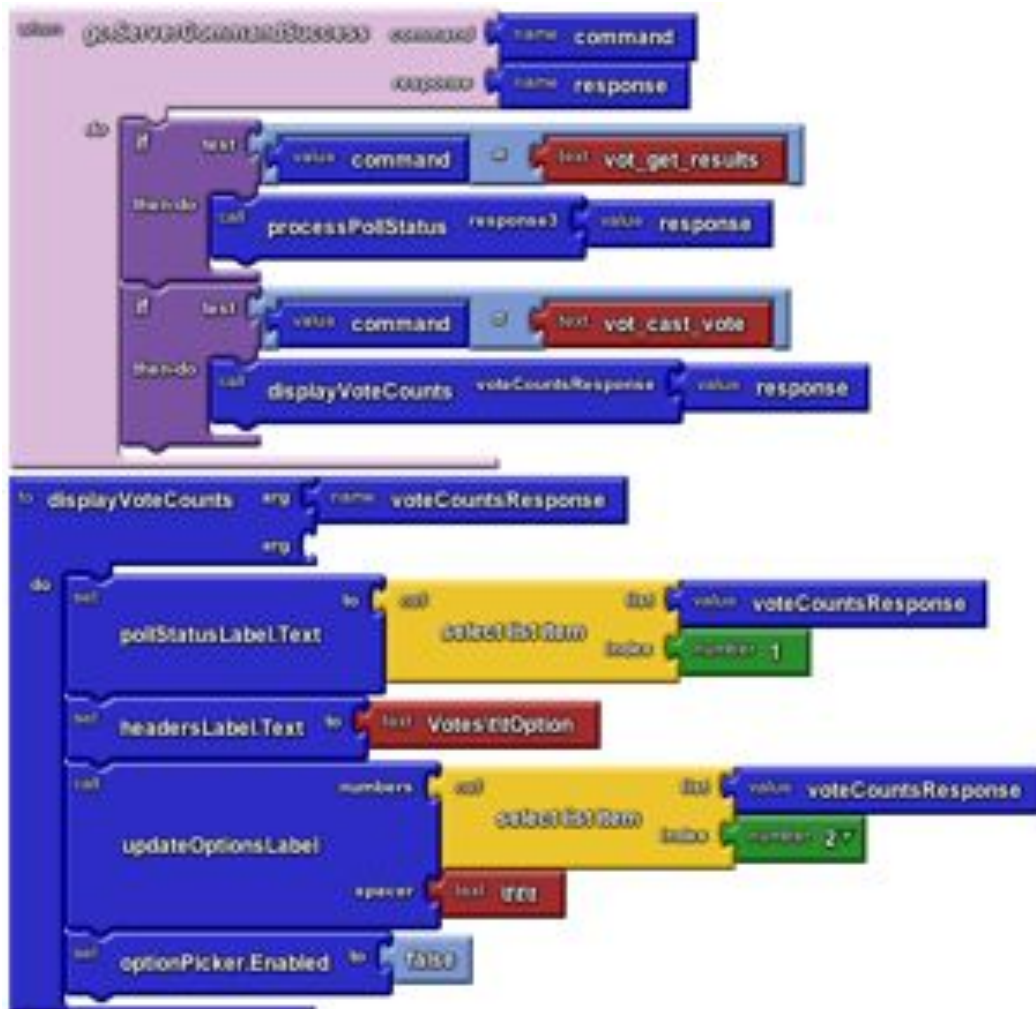


Figure 3-10: The `ServerCommandSuccess` event handler and the blocks used to update the user interface with the response from a cast vote command. All server commands return App Inventor lists as their response.

Chapter 4

System Overview

This chapter is an overview of the implementation and design decisions of the multiplayer framework. It first discusses the game server request API, extensions, custom server modules, data models and testing strategy. Then, the GameClient component's properties, method calls and events are presented.

4.1 The Game Server

The game server is implemented in Python using the App Engine SDK. It provides a set of request handlers and server commands which can be called by the GameClient component. Testing of the server is done using the Nose GAE plugin for the Nose unit testing system. Nose GAE emulates the App Engine data store on the local file system and runs tests in the restricted AppEngine runtime environment[7].

4.1.1 Data Models

The game server uses three data models to store game and player information.

- Game - Each application has one Game object in the data store. The Game is identified by the game ID property of the application's GameClient component. The game ID operates as a namespace, allowing each application to maintain its own set of GameInstances.

- **GameInstance** - A GameInstance represents a group of players participating in a single game. A Game is allowed to be the parent of any number of GameInstance children, but each GameInstance must have a unique instance ID among its siblings. Together, the game ID and instance ID uniquely describe a GameInstance object. GameInstances keep track of their current membership, the list of players who have been invited to them, their current leader, whether they are open to the public and their maximum allowable player count¹.
- **Message** - Message objects are used to send information to a player in a GameInstance. Messages can be sent using the SendMessage method call block or created directly by custom server commands. Messages contain their type, content, time of creation, and the email addresses of their recipient and sender. The content property of a Message object is a JSON string that is decoded into Python types when the Message is accessed by extensions and server commands. Lists, dictionaries, strings, booleans and numbers are all acceptable content types for the contents of a message. When the GameClient component receives a message it automatically converts the content into Yail-Lists² to return to the GotMessage event. Messages are not actively delivered to their recipient, the player must first requests them using the GetMessage block.

GameInstances and Messages are implemented as Expando objects, which allows each one to have a different set of dynamic properties. These dynamic properties are automatically created when database models are placed in the database. An example of the assignment of dynamic properties to a Message object is shown

¹This is an optional field. All GameInstances are initially created with no maximum player count. It can later be set with a server command by the game creator. Setting the maximum number of players in an instance causes it to become full if the number of players that have joined it reaches the maximum. An instance that is full will disallow further players from joining and keep it from appearing in the invited games list of all players.

²Dictionaries are converted into a list of lists of two items. Each sublist represents one entry in the dictionary and has the key of the entry as its first item and the value of the entry as its second item. This conversion will change once support for dictionaries is added to AppInventor.

below in the Make New Poll command of the custom voting module. Each field is stored in the App Engine database when the poll is “put” and can be accessed later by any server operation.

4.1: Adding dynamic properties to a Message object. (Excerpt from A.13)

```
1 poll = Message(parent = instance, sender = player,
2               msg_type = 'poll', recipient = '')
3 poll.votes = [0] * size
4 poll.open = True
5 poll.voters = ['']
6 poll.put()
```

Using dynamic properties allows custom server modules and extensions to add functionality to GameInstance and Message objects without modifying the request handlers or other server code. Messages that are retrieved using the GetMessages command from the GameClient component are returned without any dynamic properties in order to standardize the format of returned messages.

4.1.2 Request API

The server defines nine request handlers which can be called by the GameClient component. The code that implements the request handlers can be found in Code Listing A.1. Each request handler follows the same execution pattern:

1. Request variables are retrieved from the POST request object created by the GameClient component. Every request accepts a game ID, instance ID and player ID in addition to other variables specific to the request. The player ID must include the email address of the requesting player³.
2. The server starts a transaction and executes the request with the retrieved parameters. If an uncaught error is encountered during execution, the transaction will revert its changes and return an error message to the GameClient.

³To make it easier to supply email addresses, the game server will automatically use a regular expression to parse the email address from a player ID. This means that email addresses formatted with a display name (e.g. “Bill Magnuson” (billmag@mit.edu)) can be submitted without issue.

3. The transaction returns the database model specified by the game ID and instance ID of the request along with the result of the request⁴.
4. An `OperationResponse` object is created with the result of the transaction. The current state of the targeted instance is also included with the `OperationResponse` in order to keep the `GameClient` up to date on changes to the leader and membership of the instance.
5. The `OperationResponse` is converted to JSON and returned to the requester.

The `GameClient` component then processes the information in the `OperationResponse` and triggers appropriate events.

The nine request handlers and the values they return to the `GameClient` component are as follows:

- `GetInstanceLists` Returns three lists of instance IDs:
 - Public - Instances that do not require invitations to join.
 - Joined - Instances that the player has previously joined or created.
 - Invited - Instances that the player was invited to but has not yet joined.

The Public and Invited lists only include instances that currently have less players than the maximum number allowed to join that instance. Thus, any instance in these lists can be joined by the requesting player at the time of the request.

- `NewInstance` - Creates a new instance of the game specified by the game ID argument. The instance ID parameter is used as the first candidate for the ID of the new `GameInstance` object. However, if the requested ID is already assigned to a `GameInstance`, `NewInstance` will append a number on the end of the candidate instance id to make it unique. Finally, the instance id of the created `GameInstance` is returned to the `GameClient` so that it can use it as a POST variable in future requests.

⁴The result of a game server transaction is always formatted as a Python dictionary

- InvitePlayer - Invites a new player to an instance. Once a player is invited to an instance, the instance ID will appear in their Invited instance list until the game reaches its maximum membership. Non-public games can only be joined by players who have been previously invited to them. If a player is invited to a public game, it will show up in both their Public and Invited lists.
- JoinInstance - Attempts to add the requesting player to the specified instance. If the player is already in the instance, the request returns the instance ID parameter in the OperationResponse. If he or she is not in the instance, but are allowed to join because the instance is public or the player has been invited, the player is added to the requested instance and the instance ID of the joined instance is returned.

If a player attempts to join an instance that he or she is not allowed to join, an error is raised and the request is aborted. The GameClient component will then trigger a WebServiceError event with an error message describing the reason that the player was unable to join the requested instance.

- LeaveInstance - Removes the requesting player from an instance. If the requesting player is the leader of the instance, leadership is transferred to the player that has been in the game the longest. If the player is the last member of the instance, no new leader can be assigned and thus the instance is closed so that no one may join it in the future.
- GetMessages - Runs a database operation to find Message objects sent to the requesting player that match the search parameters provided by the GameClient:
 - Type - The string value used as the type when the Message was created. If the specified type is the empty string, Messages of any type are returned.
 - Count - The maximum number of Messages to return at once.

- Time - The time stamp of the most recently received Message of the requested type. This is automatically kept track of by the GameClient for each type. Every Message returned by GetMessages is guaranteed to have a creation time that is more recent than this parameter.
- NewMessage - Sends a new message to a list of recipients. This request allows the requester to define the type, content and recipients of the message. The content can be any JSON string⁵. If multiple recipients are provided, a message is created for each of them. A player may also choose to omit the recipients list or send an empty list. With empty recipients, the message is considered public and can be requested by any player in the instance.
- SetLeader - Sets the leader of a GameInstance to a new player. If a player makes this request while they are not the leader, no change is made and the request will provide a return value that indicates the lead change failed. If the request succeeds, the email address of the new leader is returned to the GameClient to confirm that the new assignment was successful. When a new instance is created by a player, he or she automatically becomes the leader of the game. The current leader is included in every OperationResponse to the GameClient component so that the game learns of leader changes as soon as possible.

The significance of a leader is determined by the game designer. In some cases, the creator remains the leader forever and serves almost no other purpose. Other games may pass leadership every time a turn is completed or at other transition points and rely on the current leader for a large part of the game.

- ServerCommand - Server commands are a class of operations that are narrowly useful for application creators. A server command request must include a command key and a list of arguments to pass to the server command.

⁵The GameClient component automatically converts App Inventor lists into JSON when messages are created.

Generic handling of server commands allows App Inventor programs to execute user defined procedures on the server without making changes to the GameClient component. This is an important feature because users of App Inventor are not able to make changes to the GameClient component in order to support requests that are specific to games they are creating.

Server commands are used to invoke both built in server commands and commands defined in custom server modules. Built in server commands include a set of server commands that can modify properties on GameInstances and extensions. Custom modules are groups of commands that have been built for a specific purpose. An example of a custom server module is seen in Chapter 3 for the polling application. Extensions and custom modules are each explained in more detail below.

Server commands return a response value with the same restrictions as the contents of a message. When a GameClient component receives a ServerCommand response, it triggers an event with the command key and the response as arguments.

In addition to their stated return values, NewInstance, JoinInstance and LeaveInstance return the same instance lists returned by the GetInstanceLists request. This allows the GameClient component to keep its instance lists up to date without making explicit calls to GetInstanceLists.

4.1.3 Extensions

Extensions are collections of server commands that provide generic functionality for use by custom server modules. The commands in a server module can be accessed from App Inventor programs by making ServerCommand requests or they can be directly called by custom modules. Two example extensions are provided in the default game server:

- Scoreboard - Stores a score for each player in an instance. Players can modify and retrieve individual scores using server commands. If a player requests

the entire scoreboard, it is formatted into a nested list that can be easily formatted for display with a ListPicker component before being returned. Custom server modules can use the scoreboard to keep track of the scores of the players in their instance.

- **Card Game** - Deals cards and keeps track of player's hands for an instance. The deck used can be set by any game that uses it, however, the default is the standard 52 card Anglo-American deck. The Card Game extension implements server commands to deal cards to all players, draw cards from the deck, pass cards to another player, discard cards from one's hand and reshuffle the deck. Each time a change is made to a player's hand, Card Game automatically sends that player's new hand to them as a Message. This allows a player to receive the current state of their hand by making a GetMessages request.

These extensions are used in the MoBullsAndCows and Androids to Androids games shown in Chapter 5.

4.1.4 Custom Modules

Custom modules are collections of server commands, which are created to provide advanced functionality to App Inventor programs. Custom server commands can implement their own game logic, utilize third party Python libraries, call extensions directly and access database models.

Implementing operations in custom server modules is an important mechanism for moving application functionality from the Android client program to the game server. Custom server modules allow application creators to implement commands to achieve any of the following outcomes:

- Simplifying game logic by grouping operations into a single server command. Custom server commands are executed using the ServerCommand request handler, which means that they are completed inside of a transaction.

- Shifting computationally intensive operations to the server. One example use case is creating a chess game with a computer player. Performing computations for a high quality chess AI requires a large amount of computation which would hog the resources of a mobile phone, but could be easily computed on an external server.
- Invoking third party libraries to access external data sources, utilize web API's or read RSS feeds. An example of a program utilizing a third party data source is seen in the Amazon example in chapter 5.
- Modifying database models to store more information relevant to the game being created. This was done in the Polling application to store special poll properties in Message objects.
- Helping students in assignments or class projects by having a member of the course staff implement server commands for students to use.

Each custom server command accepts the same three parameters: a database model, the email address of the requesting player, and a list of arguments. The database model and email address of the requesting player are automatically provided by the `ServerCommand` request handler from the game ID, instance ID and player ID POST variables. The arguments parameter is a variable length list of arguments to the command itself. The expected order and makeup of arguments vary across server commands. An identical method signature is required for server commands so that the `ServerCommand` request handler can successfully execute them without knowing the expected format of the arguments list.

Custom modules are enabled on game servers by registering their available commands in the `ServerCommand` dictionary. Registration is done automatically when the server starts if the server commands are added to a special custom command dictionary located in the Custom Modules folder of the game server. The custom command dictionary for the default game server is shown in Code Listing ??.

4.1.5 Game Server Testing

Game server testing is done using the Nose unit testing system[6]. Nose works by identifying test functions and executing them one at a time. Nose also provides two plugins that implement functionality to test Google App Engine servers:

- WebTest - WebTest starts the game server and emulates the functioning of the request handlers by accepting POST and GET commands, executing the requested transactions and returning `OperationResponse` objects. Test cases can then inspect the returned `OperationResponse` objects.
- NoseGAE - NoseGAE runs test cases in the limited App Engine runtime using a mock data store on the local machine. Combined with WebTest, NoseGAE allows unit tests to simulate POST requests as they would come from the `GameClient` and later inspect the state of the database to confirm that the correct changes have been made.

4.2 The Game Client Component

Components are the primary functional abstraction in App Inventor. Just as JAVA coders include libraries in source files to gain access to the library functionality, App Inventor users add components to their projects in order to gain access to new block drawers.

Components are implemented in JAVA and utilize libraries from both Sun's JDK and the Android SDK to perform actions during the execution of an application. Blocks are automatically created for each component by scanning its source file's public methods for JAVA annotations.

The `GameClient` component is implemented to interface with the game server. The code for the `GameClient` component and its utility classes is available in Appendix B.

4.2.1 Properties

The GameClient component provides access to nine properties:

- **GameId** - The ID for this game. The game ID can only be set in the application designer. This emphasizes that each GameClient component should target a single Game object and that the game ID should be a permanent property of the program.
- **InstanceId** - The ID of the current instance that the player is participating in. Whenever a player joins, leaves or creates a new instance, this value changes. No setter is available for the instance ID. Instead, the SetInstance method call block must be used. This emphasizes the fact that changing the instance ID of a GameClient component requires successful completion of the JoinInstance server request and cannot simply be changed on the client.
- **InvitedInstances, JoinedInstances and PublicInstances** - Each of these provide the most recently received lists for the requested instance type. LeaveInstance, SetInstance, GetInstanceLists and MakeNewInstance requests update all three instance lists when they return successfully.
- **Leader** - The most recently received leader for the current instance. Every successful server request includes the current leader in the OperationResponse. This means that the Leader property can change as a side effect of making any request.
- **Players** - The list of player that have joined the current instance and not yet left. Note that a player does not need to have a game actively open to appear in this list, they only need to have once joined the game. The Players list is also sent with every OperationResponse and evaluated for changes.
- **ServerUrl** - The web address of the game server. Like the game ID, this can only be set in the designer to disallow an application from accidentally changing the server URL partway through a game.

- `UserEmailAddress` - Provides the Google account address that was initially used to register the phone. This is the only property with a setter block, although, it should only be used in testing situations⁶. If the setter is used, the `UserEmailAddress` should be set when a program first opens because the `GameClient` component has not been designed to handle the `UserEmailAddress` changing during program operation. Additionally, allowing a player to set their own email address could result in players spoofing their identity and interfering with games.

4.2.2 Method Calls

The `GameClient` component defines one method call block for each request handler in the server API presented in Section 4.1.2. Each method call block calls a function in the `GameClient` component. This starts an asynchronous operation that completes the server request in a separate thread and triggers events after it returns. Performing the request in a separate thread allows the program to remain responsive while server requests are completing.

Each server request automatically includes the `GameId`, `InstanceId` and `UserEmailAddress` properties in the POST variables. If a server request handler requires additional parameters, the method call blocks for those requests will include sockets for each of the remaining parameters. These parameters are defined by the application creator by plugging values into the argument sockets. The blocks editor compiler performs checks at packaging time to ensure that all argument sockets have been filled with the appropriate block type. This helps keep new users from making mistakes when using method call blocks.

When a server request returns, it automatically decodes the `OperationResponse` JSON object and checks the instance ID, leader and players fields. If the instance ID does not match the current ID and the operation is not expected to result in a change of the instance ID, the response is ignored. This is done to eliminate slow

⁶This could be necessary if an emulator fails to retrieve a registered email address or an application creator needs to use the same device to simulate multiple players.

and out of order server requests that return after the user has joined a new instance. If the leader changes, the Leader property is updated and a LeaderChanged event is raised. Similarly, the players list is compared to the current Players property and if the received list is different, the Players property is updated and the appropriate PlayerLeft and PlayerJoined events will trigger.

After the OperationResponse has been checked, the transaction response in the OperationResponse is extracted and returned to the function's asynchronous callback. When a GetMessages or ServerCommand call returns, it retrieves the response contents and triggers either GotMessage or ServerCommandReturned events. If a GetMessages request returns multiple messages, the GotMessage event handler will fire once for each message.

When a request is completed, it triggers a FunctionCompleted event with its function name as the only argument. This allows program creators to perform actions when calls such as GetMessages complete successfully, but no new messages are received. An example of this is seen in the example Bulletin Board application's message reading loop in Section ??.

4.2.3 Events

Events are triggered automatically by the GameClient component when special conditions are satisfied. Many of these events have already been mentioned above in the context of property changes or returning method calls. The events that cause each of the 14 GameClient event handlers are as follows:

- FunctionCompleted - A function completed successfully. This is called with the name of the function as the only argument.
- GotMessage - A message was received after a call to GetMessages. Each received message includes its type, sender and contents.
- Initialize - Triggers automatically at program startup. This should not be used in the GameClient except to set the UserEmailAddress when necessary.

- `InstanceIdChanged` - A call to `SetInstance`, `MakeNewInstance` or `LeaveInstance` completed successfully and the `InstanceId` property changed as a result. The new value of the `InstanceId` property is provided as an argument.
- `Invited` - A request that updated the instance lists completed successfully and the player has been invited to a new instance. The ID of the instance the player was invited to is passed to the event handler.
- `NewLeader` - The leader of the current instance has changed. This could be the result a player (including the current one) calling `SetLeader` or a `ServerCommand` changing the leader field of the current `GameInstance` object.
- `NewInstanceMade` - A `MakeNewInstance` request completed successfully. Like `InstanceIdChanged`, the event handler provides the current value of the `InstanceId` property as its only argument.
- `PlayerJoined` and `Player Left` - The `Players` property has changed due to a player entering or leaving the instance. Their email address is provided as an argument to the appropriate event handler. These handlers can trigger multiple times on a single request if more than one player enters or leaves a game.
- `ServerCommandFailure` - A `ServerCommand` failed. The event handler provides the command key and the original arguments to the `ServerCommand`.
- `ServerCommandSuccess` - A `ServerCommand` succeeded. Passes the command key and the `ServerCommand` response as arguments to the event. The command key is provided so that the program knows how to handle the response correctly.
- `UserEmailAddressSet` - The user email address property has been successfully set to a non-empty value. This event should be used to initialize any web service functions. The `UserEmailAddress` will attempt to set itself to the

Google account registered with the phone. If this fails, the `UserEmailAddress` must be set with the property setter.

- Errors - The `GameClient` component triggers two different error events:
 - Info - Triggered when a player attempts to perform an action with improper arguments. Info events should generally only happen during application development as the application creator should be able to keep them from happening in finished programs.
 - Web Service Error - Caused by a network failure or server error. These errors are raised with the name of the method call that caused the error and a text value containing an error message. If the `WebServiceError` occurs because of an aborted server request, the message will be the text of the server exception. Otherwise, the message is a summary of the network failure that occurred.

With these two pieces, the multiplayer framework enables a wide variety of applications and games to be built. The next chapter presents four example applications and discusses their designs to demonstrate the diversity of possibility when using the framework.

Chapter 5

Further Examples

This chapter presents four example programs that make use of the game framework in different ways. The first, Bulletin Board, is implemented using the unmodified game server and requires no use of server commands. It functions as a multi-user online message board. Users can create or join different boards and leave messages for others to see.

The second application is a reimplementaion of a program made by David Wolber, one of the professors in the App Inventor pilot program. With a custom server module, it accesses Amazon's e-commerce web service to look up books by keyword or ISBN. By using the game framework, the entire program can be made with under 50 blocks and a 25-line server module. This program shows the potential for games to utilize online data providers and other web services.

The next application, MoBulls and Cows, is a Bulls and Cows¹ game variant which depends on the game server to perform game logic and keep score. MoBulls and Cows uses the Scoreboard extension to keep track of the high and average scores of all players so that users can compete against each other.

The final application is a multiplayer card game called Androids to Androids that uses a custom server module, and both the Card Game and Scoreboard server extensions. The game uses a state machine that permits a player to close the pro-

¹Bulls and Cows is a game similar to the popular game Mastermind. Full rules and a description of the game can be found at http://en.wikipedia.org/wiki/Bulls_and_cows.

gram in the middle of a game and return to it later without losing his or her place. This also allows players to simultaneously participate in multiple instances of Androids to Androids with different groups of players.

5.1 Bulletin Board

The Bulletin Board application uses game instances as separate bulletin boards that contain messages posted by users. Figure 5-1 shows the interface for viewing the FreeFood bulletin board. Users select the board they would like to view from a ListPicker that includes all public bulletin boards. After joining, the player can see the last 10 messages posted to the bulletin board and is able to post his or her own messages for others to see. Every operation required to build Bulletin Board is included in the default game server.

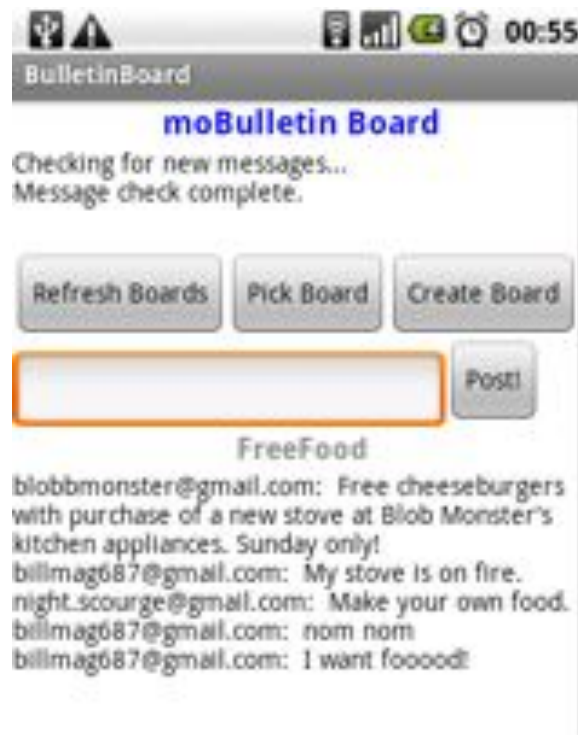


Figure 5-1: Viewing the FreeFood bulletin board.

When a user opens Bulletin Board, a `GetInstanceLists` request is made in order to populate the list of currently available bulletin boards. While this is happening,

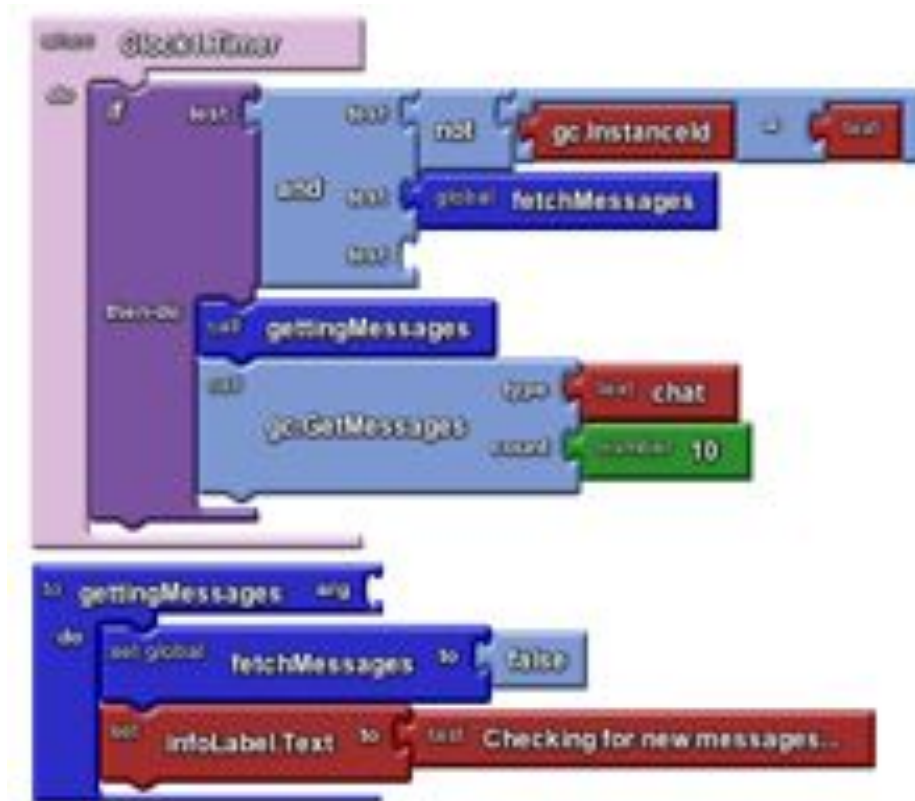


Figure 5-2: The Bulletin Board message reading loop. Bulletin Board uses a Clock component to make message requests every 10 seconds.

the “Pick Board” button is disabled. Once `GetInstanceLists` returns, the user is informed that the list of bulletin boards has been refreshed and the button becomes enabled. At any time after this, the user may select a new bulletin board to view and replace the currently displayed messages.

To fetch new messages automatically, Bulletin Board uses a Clock component which triggers a Timer event every 10 seconds. The handler for the Timer event retrieves new messages from the server. The event handler is shown in Figure 5-2. The body of the event handler uses a boolean value to make sure that a second `GetMessages` call is not made before the first one completes. This is done by setting the variable to false when starting the request and only returning it to a value of true when `GetMessages` returns.

When a new message is received, its sender and contents are formatted into a display string and added to the top of the list of posted messages. The message-

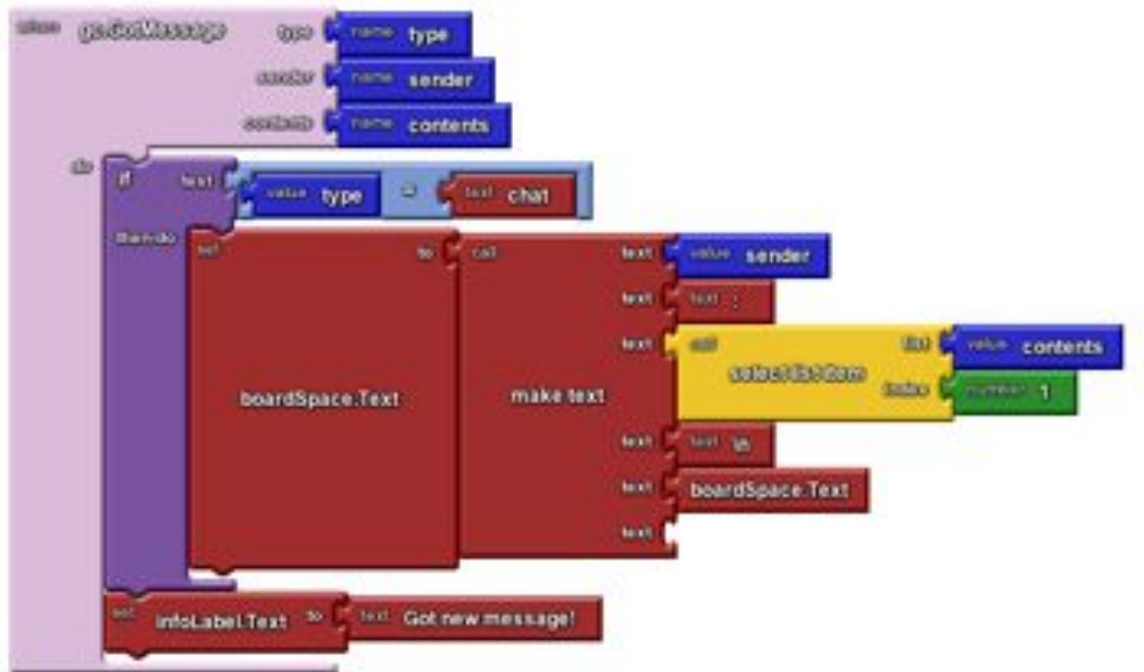


Figure 5-3: The event handler for new messages. When a new message is received, its sender and content are merged into a single text value and added to the bulletin board display.

reading loop and GotMessage handler (shown in Figure 5-3) rely on the assumption that the GameClient component will only request new messages when a call to GetMessages is made.

5.2 MoBulls and Cows

MoBulls and Cows is a version of the classic pen and paper game Bulls and Cows, which uses a custom server module with two commands. Each time a player opens MoBulls and Cows, a new game is started by making a server request. At the beginning of a new game, the server randomly chooses a sequence of four colors from a set of six. Each selected colors appears only once in the solution. This solution sequence is only known by the server.

The player then attempts to guess on the correct sequence. After each attempt, they are informed of how many “bulls” and “cows” are in their guess. A “bull”



Figure 5-4: The MoBulls and Cows game after submitting the correct sequence. Each guess is sent to the server when the player hits the “Submit Guess” button. The server then checks the guess against the correct sequence and returns the number of “bulls” and “cows”. If the player wins, the server will update his or her score statistics and send them back with their final score.

represents a correctly guessed color in the correct position and a “cow” indicates that a color in the guess is correct, but it is in the wrong position.

The player begins with a starting score of 96². After each guess is made, two points are deducted for each item in the guess that has a color not appearing in the solution and one point is deducted for a correct color that is in the wrong spot (a cow). No points are deducted for a bull. If a player does not determine the correct sequence before they run out of guesses, they lose the game and must start over.

To submit a guess, a player chooses a color for each of the four places in the solution. MoBulls and Cows uses ListPicker components that change their background color depending on the color chosen to display the current guess. These can be seen to the left of the “Submit Guess” button in Figure 5-4. When a player

²This starting score is chosen so that a guess with zero “bulls” and “cows” on every turn would result in a score of zero.

hits Submit, the game checks to make sure that the player is not accidentally repeating a guess and then sends it to the server. The Click handler for the “Submit Guess” button is shown in Figure 5-5

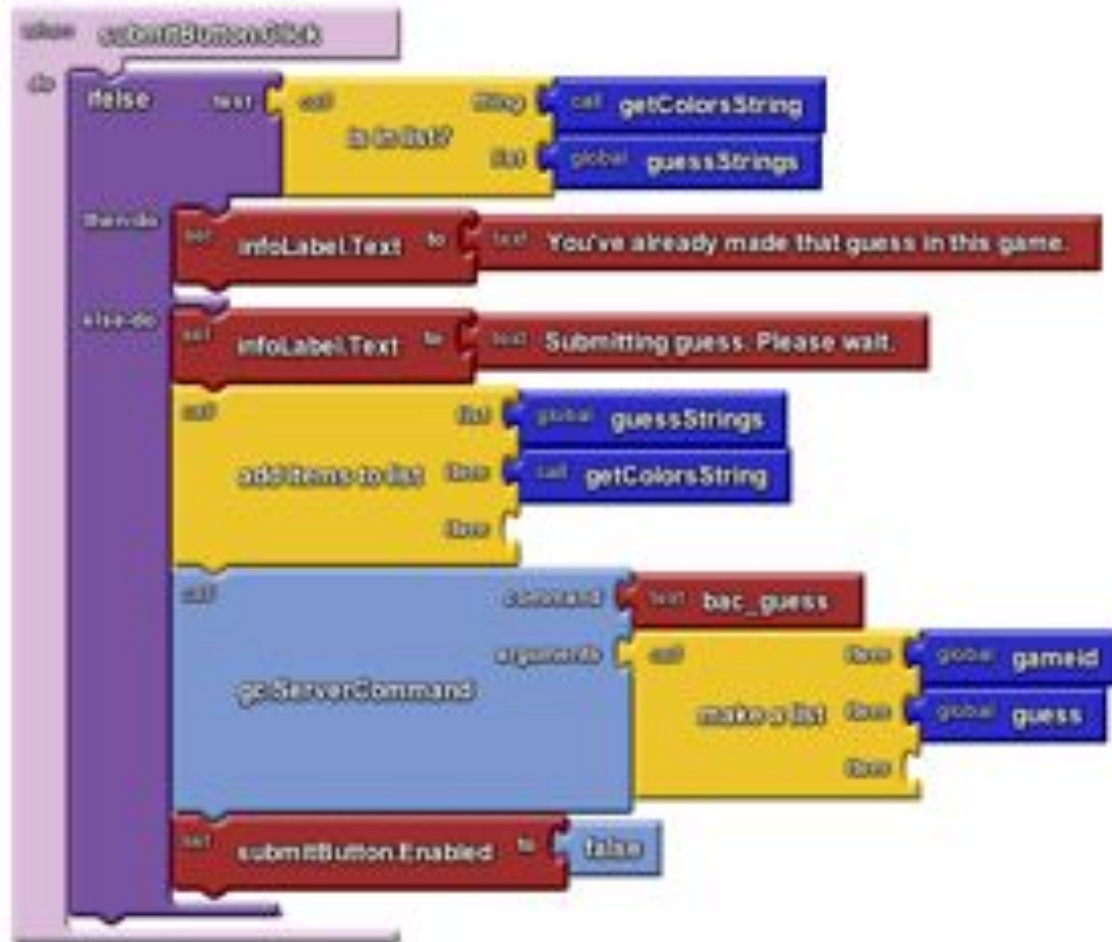
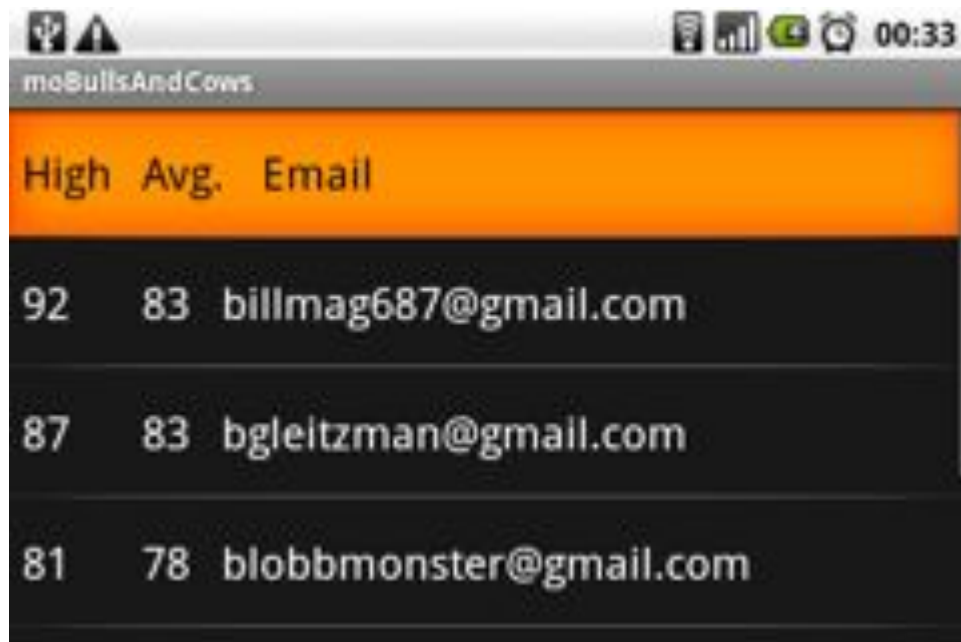


Figure 5-5: The blocks used to submit a new guess to the game server. The game first checks to make sure that the player has not previously tried the same guess and then submits it to server.

The Bulls and Cows custom server module then processes the guess, determines the number of “bulls” and “cows”, and adjusts the player’s score accordingly. If they have guessed the correct solution, they are awarded their final score and the current game scoreboard is sent back with the server response.

In order to provide a game-wide scoreboard that incorporates all users of the program, the same game instance is used for all players. At any time, players can view the scoreboard by clicking on the “View High Scores” ListPicker. This will

show a screen like the one shown in Figure 5-6 with the high and average scores of all players in the game.



High	Avg.	Email
92	83	billmag687@gmail.com
87	83	bgleitzman@gmail.com
81	78	blobbmonster@gmail.com

Figure 5-6: Viewing the scoreboard for MoBulls and Cows. High scores and statistics are kept track of by the server in a scoreboard that is stored with the game instance. Programs can request the scoreboard with a server command.

If a server command fails due to network problems, MoBulls and Cows will notify the user of the failed attempt and automatically retry up to a maximum of five times. With this approach, it is important to guard against scorekeeping issues that could arise from the client automatically submitting the same guess more than once. To avoid this problem, the Submit Guess function caches the most recent guess and returned value. If the client submits the same guess repeatedly, the custom module does not modify the database and instead replies with the cached return value. This caching scheme requires only six lines of Python code in the server module (see Code Listing A.12).

5.3 Amazon

Amazon is a simple program for looking up books in Amazon's listings. It demonstrates a mobile application created with a few dozen blocks and a short server module that accesses external online resources.



Figure 5-7: The Amazon program after looking up a book by keyword. The game server accesses the Amazon E-Commerce Services to perform a query for the keyword and returns any books it finds to the program.

Users operate the Amazon program by entering a book keyword or an ISBN into a text input box and clicking the search button. Then, a server command is made, which uses a custom server module to access the Amazon E-Commerce Services API and perform a search of Amazon.com's book inventory.

The returned results include each book's title, price on Amazon.com and the Amazon Standard Identification Number (ASIN). In order to keep the program logic simple, the server commands for searches by ISBN and keyword return their

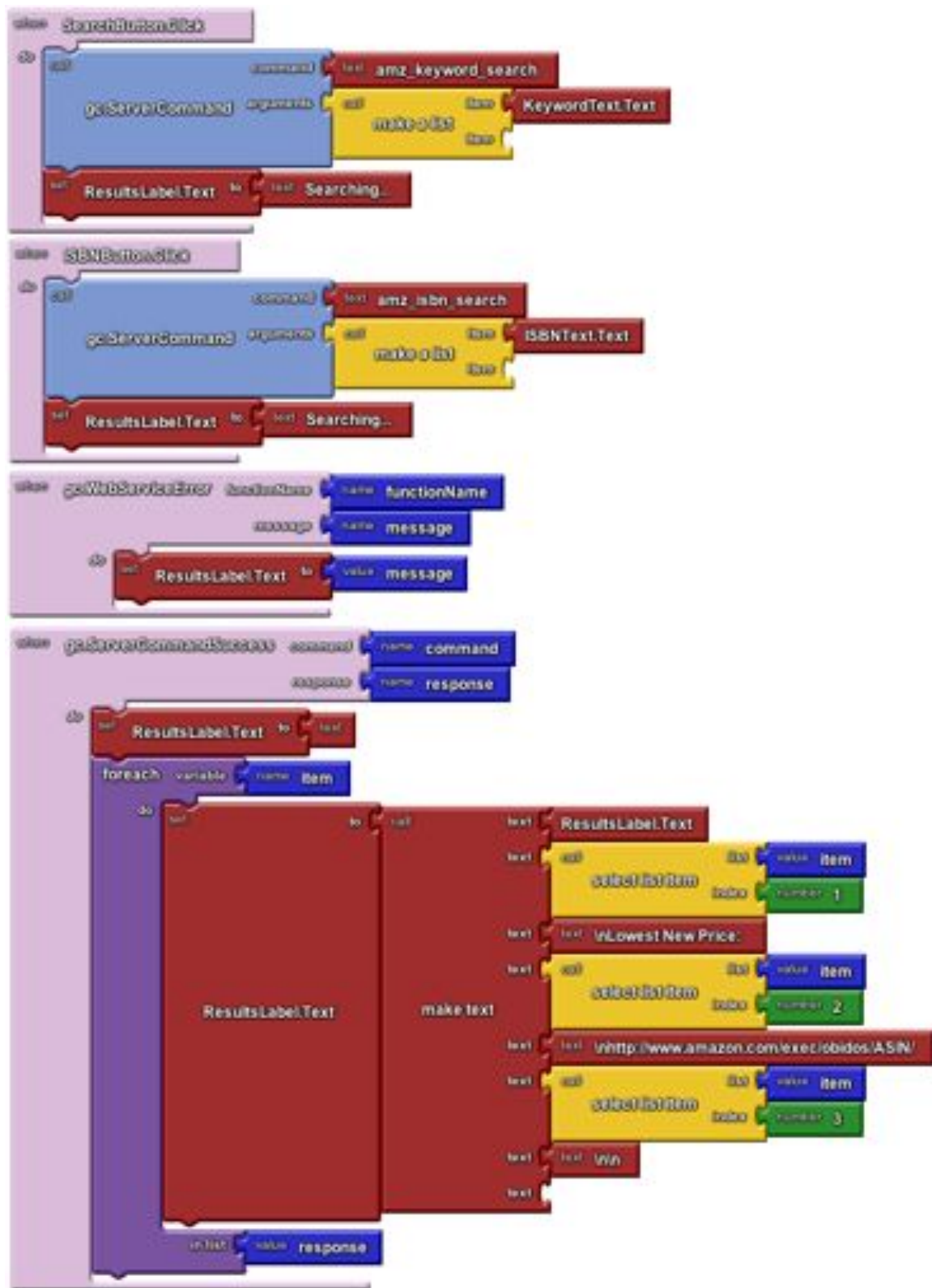


Figure 5-8: The entire blocks workspace for the Amazon program. By returning book information in the same format for both keyword and ISBN searches, all server command responses can be handled with the same blocks.

results in the same data format. This allows the ServerCommandReturned event handler to treat all server response lists identically. The entire blocks workspace of the Amazon program can be seen in Figure 5-8.

Access to the E-Commerce Services is done with a third-party Python library that accepts keywords or ISBNs and returns Python iterators of book objects. With this library, the server module only needs to format the results from the iterator into lists that can be returned to App Inventor. The resulting custom server module is less than 25 lines of Python code. The code for a keyword search is shown below (some documentation has been omitted for brevity).

5.1: Server code required to perform a search by keyword. (Excerpt from A.10)

```
1 def amazon_by_keyword(keyword):
2     """ Use the ecs library to search for books by keyword.
3
4     Returns:
5     A list of three item lists. Each sublist represents
6     a result and includes the book title, its lowest found
7     price and its ASIN number.
8     """
9     ecs.setLicenseKey(license_key)
10    ecs.setSecretKey(secret_key)
11    ecs.setLocale('us')
12
13    books = ecs.ItemSearch(keyword, SearchIndex='Books', ResponseGroup='
    Medium')
14    return format_output(books)
15
16 def format_output(books):
17     """ Return a formatted output list from an iterator returned
18     by the ecs library. """
19
20     size = min(len(books), return_limit)
21     return [[books[i].Title, get_amount(books[i]), books[i].ASIN]
22             for i in xrange(size)]
23
24 def get_amount(book):
25     """ Return the lowest price found or 'Not found.' if none exists."""
26     try:
27         if book.OfferSummary and book.OfferSummary.LowestNewPrice:
28             return book.OfferSummary.LowestNewPrice.FormattedPrice
29     except:
30         return 'Not found.'
```

5.4 Androids to Androids

Androids to Androids is a multiplayer card game played by groups of three or more players³. Androids to Androids uses two different decks of cards:

- Noun Cards - Contain the name of a person, place or thing. A player's hand consists of seven noun cards at all times.
- Adjective Cards - Contain a description word. At the beginning of each round, an adjective card is chosen and displayed to all players. Players then choose the noun card that they think is the best match for the round's adjective and submit it to the leader.

When players first open the Androids to Androids program, they must either join a game or make their own. If a player creates a new game, that player is automatically the first leader of the game and must wait until at least two other players join his or her game before beginning. Once a game begins, it continues in rounds until one player reaches a score of five. The winner of each round is determined by the current round leader and is awarded one point. Each new round is led by the winner of the previous round. To start the game, each player is dealt seven noun cards.

At the beginning of each round, an adjective card is chosen at random by the server and sent to each player. Every player, except the leader, then chooses a noun card from his or her hand and submits it to the leader for review. When a player submits a card, the server will send it to the leader of the round and replenish the player's hand with a randomly chosen noun card from the deck. The server keeps track of the hands of all players in the game using the built in card game server extension.

To end a round, the leader selects a winner from the set of submitted noun cards. Players usually submit cards that apply thematically to the characteristic card for the round, although, the leader of a round is allowed to use any selection

³Androids to Androids is a variant of the party game Apples to Apples published by Mattel.

criteria they wish in determining the winner. Learning the selection preferences of other players is very important to apply proper strategy during the course of a game.



Figure 5-9: The user interface after two additional players have joined a new game. The “Start Game” button is activated and will send a server command to begin the game if clicked.

5.4.1 Round Numbers

Games that proceed in rounds, such as Androids to Androids, present a unique design challenge because in order to work properly, all users must be in sync throughout the course of the game. To help solve this problem, the client keeps track of its view of the round number and includes it as an argument to each server command. If the locally stored value is ever less than the current value held on the server, the server knows that the client has fallen behind and provides the current



Figure 5-10: Androids to Androids after submitting a card. Players choose cards from their hand using a ListPicker component and submit them to the leader. In response to the card submission, the server will replenish the player's hand with a new card drawn randomly from the deck and send it back to them.

round's information. If this catch-up response is received, the client will ignore the action made by the player and instead re-sync with the server.

The code for submitting a card is shown below. The first check done by the server command is to make sure that the card has been submitted for the correct round. The user interface after a successful submission is shown in Figure 5-10.

5.2: The custom server command for submitting a noun card to the leader. (Excerpt from A.11)

```
1 def submit_card_command(instance, player, arguments):
2     """ Submit a noun card for the current round.
3
4     Args:
5         instance: The GameInstance database model for this operation.
6         player: The player submitting the card. Cannot be the leader.
```

```

7     arguments: A two item list consisting of the round to submit this
8     card for and the card itself.
9
10    If the submission is for the wrong round, a four item list with an
11    error string as its first element will be returned. The remaining
12    elements are the player's hand, the current round and the current
13    characteristic card to respond to. No other action will be taken.
14
15    Removes the indicated card from the player's hand and adds it
16    to this round's submissions. The current submissions are sent via
17    message to all players.
18
19    The requesting player's hand will be dealt another card after
20    removing the submitted one. The updated hand will be sent to the
21    requesting player in a message and be included in the return value
22    of this command.
23
24    Returns:
25        If the submission is for the correct round, returns a three item
26        list consisting of the current round number, a list of the
27        submissions made so far by other players in this round and the
28        player's new hand.
29
30    Raises:
31        ValueError if player is the leader. The leader is not allowed to
32        submit cards.
33    """
34    if int(arguments[0]) != instance.ata_round:
35        hand = card_game.get_player_hand(instance, player)
36        return ['You tried to submit a card for the wrong round. ' +
37               'Please try again.', hand, instance.ata_round,
38               instance.ata_char_card]
39
40    if player == instance.leader:
41        raise ValueError("The leader may not submit a card.")
42
43    submission = arguments[1]
44    submissions = set_submission(instance, player, submission).values()
45    instance.create_message(player, 'ata_submissions', '',
46                           [instance.ata_round,
47                            submissions, submission]).put()
48
49    card_game.discard(instance, player, [submission], False)
50    hand = card_game.draw_cards(instance, player, 1)
51    return [instance.ata_round, submissions, hand]

```

5.4.2 Leaders

Androids to Androids makes use of a game leader to control the user interface and pass leadership of the game from player to player. Each time a request is

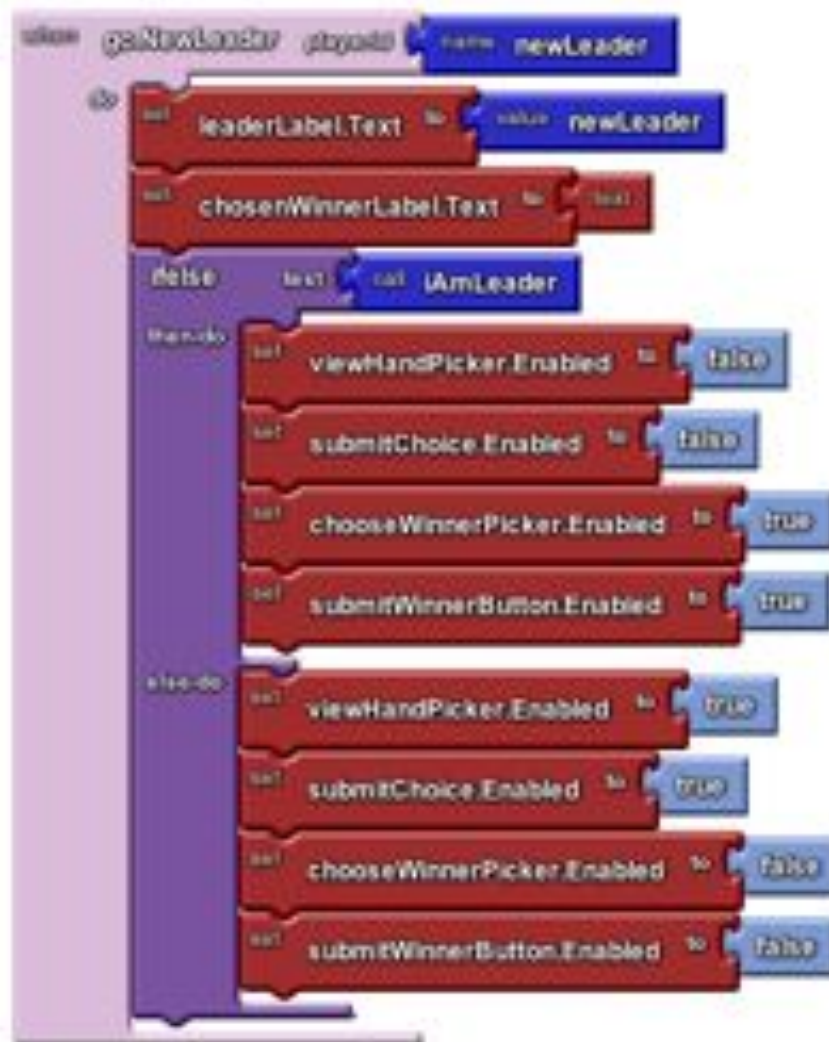


Figure 5-11: The Androids to Androids NewLeader event handler. The leader has a different set of actions than other players during a round. This event handler ensures that only necessary user interface elements are enabled for each player.

made, the server includes the current leader in its reply. On the client side, every response is checked and a NewLeader event is raised if the value of the leader parameter changes. The NewLeader event handler in Androids to Androids (see Figure 5-11) changes which parts of the user interface are enabled. This ensures that all players are performing the correct actions and not making inappropriate server calls. When a player wins a round, he or she automatically becomes the new leader and gets to choose winner of the next round.

5.4.3 Messages and Persistent State

There are five different message types used in Androids to Androids. The first four are created by the custom server module and are sent to all players in the game. The fifth type is for players' hands. Hand message are sent directly to the player from the card game extension when the Androids to Androids server module invokes its methods. The five message types are as follows:

- New Game - This message is sent to players after the leader clicks "Start Game" and the server initializes the game state. It includes the first round's characteristic card and the starting scoreboard.
- New Round - Each time the leader chooses a winning card, a new round starts with the winning player as the next leader. This message includes the new round number, the updated scoreboard, the characteristic card for the round, the winner's email and the winning card.
- Game Over - A Game Over message is sent when a player reaches a score of five. It includes the final round number, the winner's email address, the winning card and the final scoreboard. When a client receives this message, it allows the user to leave the game.
- Submissions - Submissions messages are sent whenever a player submits a card. The messages include the round number of the submission and a list of all cards submitted so far. This list is used by the leader to choose a winner

and is provided to all players to view while they wait for others to submit noun cards.

- Hand - These messages are sent whenever a player's hand is modified from dealing, discarding or drawing cards. Every time a player submits a card, a new hand message is created. Figure 5-10 shows the user interface after a player has submitted a card and their hand has been replenished.

The use of messages in Androids to Androids allows players to close Androids to Androids at any time and later rejoin a game in progress. Reloading a game is possible because the current state of the game does not rely on persisted local state and can be recreated by processing game messages. With the five message types listed above, each time the game transitions from one state to another, there is a message created containing the information required to perform the transition. Taking advantage of this, the client was built as a state machine and updates its user interface as it receives messages. If the Androids to Androids program is exited and re-opened, it requests its previous messages and replays the game internally. The game framework will make sure that messages are received by the client in the order that they were created and provides their type to the GotMessage handler so that each one is handled properly. Figure 5-12 shows the GotMessage event handler that calls the appropriate procedure for each message type.

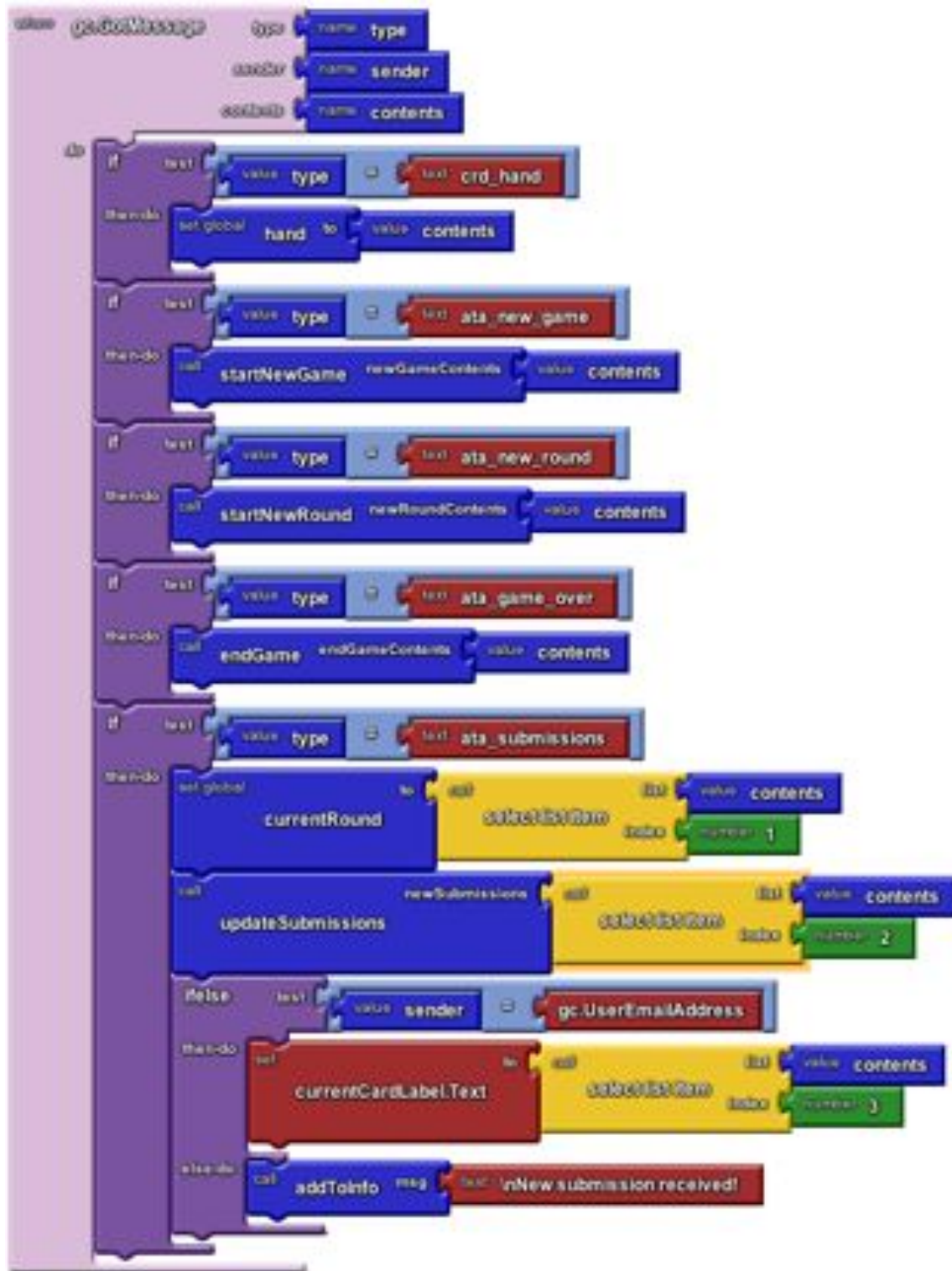


Figure 5-12: The GotMessage event handler in Androids to Androids. Depending on the type of the message received, the event handler calls another procedure with the message contents as the argument. These procedures know the specific format and ordering of the contents of their message type.

Chapter 6

Related Research

The multiplayer framework for App Inventor builds on previous work in the fields of visual programming languages, game building, teaching with computer games, and multi-user mobile phone networks. This chapter discusses the design trade-offs of related projects as they compare to the implemented characteristics of the multiplayer framework.

6.1 Visual Programming Languages

App Inventor follows in the footsteps of previous graphical programming languages, such as Scratch and StarLogoTNG, as a tool to allow users to build applications without writing code. App Inventor is unique among these languages in its targeting of the mobile phone platform and the breadth of its components.

Scratch and StarLogoTNG use blocks languages similar in appearance and function to the language used in App Inventor. Under the hood, the App Inventor blocks editor runs with a modified version of the OpenBlocks library[11]. The OpenBlocks framework is a general purpose graphical blocks language that can be configured to meet the needs of many different graphical programming projects. StarLogoTNG, which the OpenBlocks framework is based on, is an evolution of the original StarLogo[10]. StarLogo and Scratch both share influences from the Logo programming language (a dialect of Lisp developed in the late 1960's).



Figure 6-1: The user interface of StarLogoTNG. The blocks of StarLogoTNG closely resemble the blocks in App Inventor. A real time view of the running program is shown in the top right.

Scratch, which is targeted primarily toward children, includes only basic operations in its blocks language. App Inventor, on the other hand, has multiple tiers of operations in order to satisfy users with different skill levels. The design challenges of creating multiplayer games and handling asynchronous function calls push the GameClient component into a higher tier of complexity. However, the framework remains accessible to new users by simplifying its server requests into single blocks which are easy for application creators to understand.

StarLogoTNG, along with Kodu[3] and Alice[4], allow users to create games with 3D graphics. Screenshots demonstrating the blocks languages of Alice and Kodu are shown in Figures 6-2 and 6-3. Game creators define event handlers and procedures which determine the behavior and reactions of 3D characters as they move about the game's 3D environment. These languages sidestep the complexity of creating graphics, but also limit the breadth of functionality that can be included in applications. As a result, users have almost no ability to utilize resources and operations that have been predefined for the 3D environment.



Figure 6-2: A procedure in Kodu. Kodu uses when and do blocks which operate equivalently to event handlers and method calls in App Inventor. Nested when blocks are used to implement conditionals.

Graphics in App Inventor are made using 2D image sprites and a drawing canvas. Graphics and user interface design in App Inventor are still at an early stage in development. As the platform matures it will include more graphics primitives to allow users to build more visually interesting applications.

6.2 Mobile Games in Education

Many educators use games to encourage students to take an active role in their studies. Recently, mobile games have been used in education to allow students to learn through engagement with their environment. One such system, built by professors from three universities in Taiwan, uses a client/server setup similar to the multiplayer framework's design[12]. The design is aimed at allowing course instructors to provide lessons to their students, but makes no effort to give students the ability to easily customize the application on their own. Additionally,

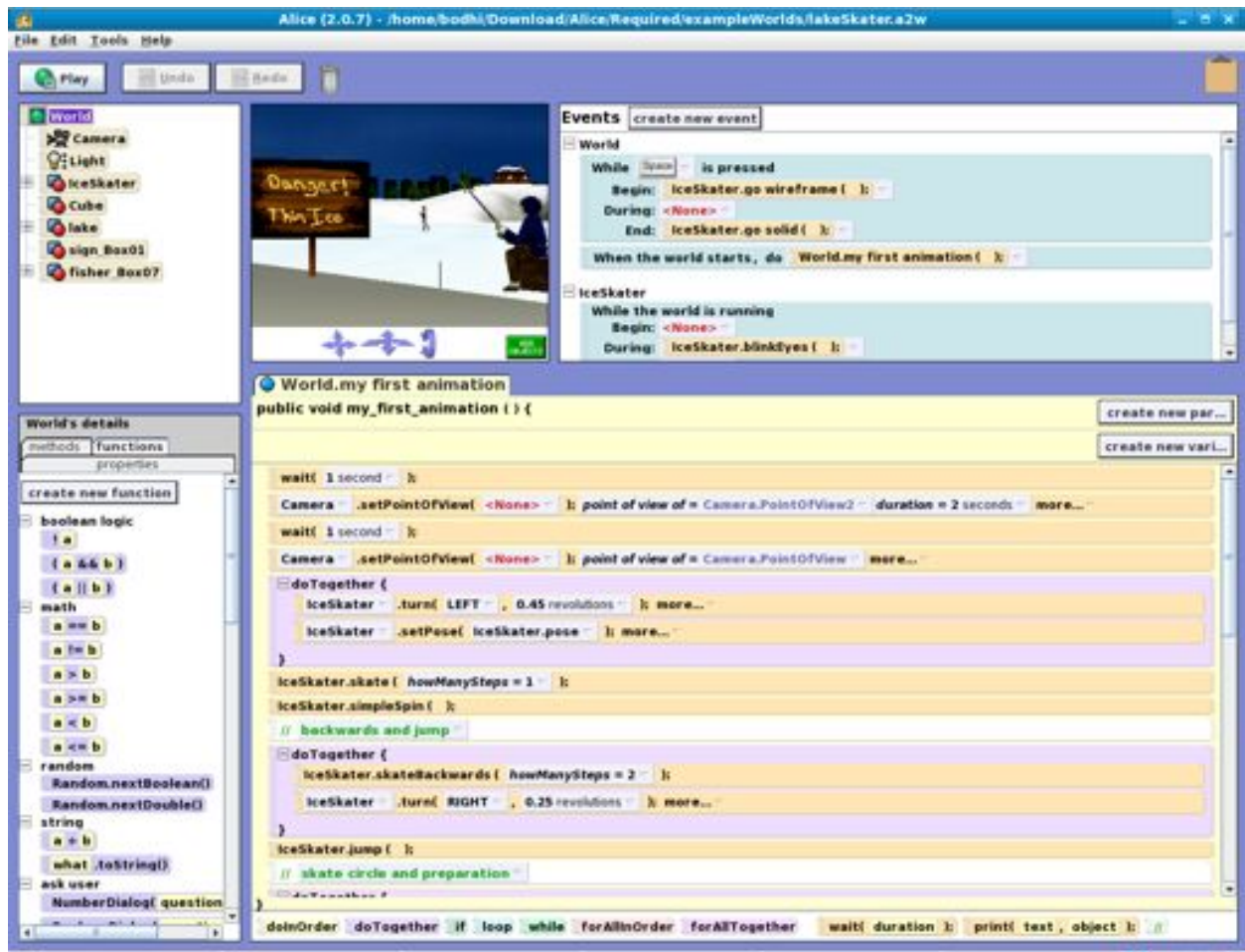


Figure 6-3: The user interface for Alice. Game creators define animations and event handlers which control the interaction of characters in a 3D environment using the guided storyboard. The display of semicolons and brackets is an optional feature of Alice that is used to transition users to Java or other written languages. The 3D animation displayed at the top of the window can be run to see the effect of changes to the storyboard during development.[4]

the lesson creator uses a rigid design structure, which limits the creativity of game creators.

The Department of Computer Science at Central Connecticut State University is also developing a new class that uses video game creation to introduce Computer Science to students. Encouraged by students' enthusiasm for computer games, but discouraged by the difficult task of teaching students the complexities of modern games, the course developers turned to mobile game design. Students are required to have a basic knowledge of Java before taking the class and are taught using the Java 2 Platform Micro Edition (J2ME)[9].

The course succeeded at teaching students to create single player games, however, most students expressed a desire to make multiplayer games by the end of the course. Unfortunately, course instructors found including multiplayer games in the curriculum to be difficult due to the the higher technical requirements of implementing data communication on mobile devices, and the breadth of pre-requisite topics[9]. Both of these issues can be solved with App Inventor and the multiplayer framework because the component system removes the ramp-up time required to teach students the J2ME platform and eliminates the need for students to implement data handling and server communication.

6.3 Alternatives to Client-Server Design

Peer to peer overlays such as the Content Addressable Network (CAN) technique and Pastry have been used as aggregation tools for distributing game state among players in massive multiplayer online games (MMOGs). One of these, SimMud, uses Pastry to avoid the large start up costs of a centralized server and handling peak loads[8]. The multiplayer framework addresses these resource problems by using the free App Engine service, which provides easy setup and automatic scaling of computing power to meet demand.

Researchers in Berlin performed an analysis of the use of the CAN technique for mobile gaming. The research, which was done in 2005, suggests that the in-

creased use of “3rd generation protocols” promotes a server-client over a peer to peer structure for mobile games, citing only cost to the user for data service subscriptions as a concern[5]. Most modern smart phones (and all Android mobile phones, which App Inventor specifically targets) are capable of using 3rd generation data networks and are generally sold with affordable, unlimited use data plans. Thus, these concerns have become dated.

One fault that remains is the high latency and poor reliability of mobile connections. In practice, Android applications often fail to successfully complete requests. These conditions make playing real-time games nearly impossible as it is very difficult to maintain real-time game state on the client[5]. This problem can be avoided by creating turn based games and building server commands that properly handle duplicate requests.

Chapter 7

Extensions

While a wide range of games and interesting applications can already be created using App Inventor, there are many areas for improvement that will streamline the game creation process and allow users to create more advanced applications. Each extension listed below discusses the improvements and changes that can be made to the multiplayer framework as the App Inventor system matures.

7.1 User Interfaces and Multiple Screens

The main challenges for current games with respect to inviting players or managing game membership are related to cumbersome user interfaces. Currently, App Inventor only supports a single screen and a small selection of user interface components. Thus, game designers must include both the game management and game playing user interfaces on the same screen. This causes game interfaces to quickly become cluttered and confusing. A new player that opens the application will be tempted to immediately start using the game playing interface before he or she has even joined a game. Application designers can currently handle this by disabling parts of the user interface to shoehorn several different modes into a single screen, but this is often very confusing for users and requires a large number of blocks to implement. Future versions of App Inventor will include more powerful and diverse user interface capabilities.

7.2 Saving Local State

Another problem for application designers is dealing with the lack of persistent local state in App Inventor applications. Currently, when an application is interrupted it loses all of its state and completely re-initializes when it is reopened. During the course of a long lasting game, it is likely that the game will be interrupted by a phone call, text message or other activity on the phone.

The game server's messages and the TinyWebDB component can both be used to persist application state on the web, but in many cases components have state that is not exposed to the user as properties. In the GameClient component this includes a dictionary of message types to receipt times that is stored as a private hashmap in the component's Java code. Similarly, storing received messages, the text on a label, or the elements of a list picker would all make dealing with game interruptions much easier.

7.3 Pushing Messages and Game State Updates

In the present design of the game framework, applications are forced to poll the game server to receive new messages or to update game state information, such as the current players or the leader. Application developers can currently maintain an up-to-date view of the game state and messages by constantly repeating server requests with a Clock component or by triggering new calls immediately after a previous one returns. Unfortunately, this approach is ineffective and costly on a mobile phone. Mobile data connections are unreliable and often require round trip times measuring in seconds just to complete a single server request. Programs must also be careful not to use too much battery life or users will be unwilling to run them.

In the future I expect the App Inventor system to support a way to push information to applications or create long-lived server connections. This would allow games to immediately become aware of new messages, lead changes, and new

players. The game framework is well suited to using this architecture because all game changes are caused by events performed by other players. If such a change was made, the method blocks for `GetInstanceLists` and `GetMessages` could be removed. Application design could then focus on what to do in response to the receipt of messages instead of trying to optimize data usage and performance by fine tuning when they should be retrieved.

Chapter 8

Contributions

The multiplayer framework for App Inventor for Android makes the following contributions:

1. Integrated a GameClient component into App Inventor for Android system which enables application developers to use the game server and other App Engine capabilities with App Inventor applications.
2. Built a game server with game management, message passing, extensions and custom module support using the Python App Engine SDK.
3. Developed four custom modules to show the integration of user created commands into the game server to implement game logic, leverage extensions, and access third party data services.
4. Created five example applications to demonstrate possible uses of the multiplayer framework.
5. Utilized the Nose GAE testing system to create a unit test suite for the game server and its modules, which runs in the Google App Engine sandbox.
6. Released the GameClient and open source game server code through the App-Inventor-for-Android Google Code project.

Appendix A

Game Server Code

This appendix includes the request handlers, database models, server extensions and custom modules written for the game server to run on AppEngine. All unit tests and third party code have been omitted. For the complete runnable server code, see the App Inventor for Android project on Google Code at: <http://code.google.com/p/appinventor-for-android/>.

Files are organized in sections according to the directory structure of the game server.

To view more documentation and download the example programs presented in this thesis please visit the App Inventor Help site at <http://sites.google.com/site/appinventorhelp/>

A.1 Game Server

A.1: server.py - The game server application file. Includes the request handlers for server requests.

```
1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #      http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
```

```

10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 Defines the request handlers for the game server. After retrieving
15 arguments from the request, all operations are run as database
16 transactions. This means that any unhandled errors encountered
17 during the operations will result in the database performing a
18 'rollback' to the state that it was in before the request was made.
19
20 All server command functions return a tuple of the database model
21 they operated on and a dictionary of results. These vary from command
22 to command, but all requests will provide their return value using an
23 OperationResponse object.
24
25 The get functions for each request handler provide a simple web form
26 to perform the operation via a web interface and will write their
27 responses as a web page. Put functions write json to the request
28 handler that can be consumed by other applications.
29
30 Throughout this module, pid is accepted as an argument. The correct
31 format for a pid is of one of the following forms:
32 'Bill Magnuson' <billmag@mit.edu>
33 billmag@mit.edu
34
35 Received pids will be parsed for the email address and only the email
36 address will be used to identify players during game
37 operations. These same rules apply to other fields which identify
38 players such as a new leader or an invitee. In general, the variable
39 name 'player' will be used to represent values that are email
40 addresses and pid is used more generally to indicate that other
41 strings are acceptable as input.
42
43 For more information about the validation done on game ids, instance
44 ids, and player ids, look to utils.py.
45 """
46
47 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
48
49 import sys
50 import logging
51 import traceback
52 import iso8601
53 import utils
54 from datetime import datetime
55 from django.utils import simplejson
56 from google.appengine.ext import webapp
57 from google.appengine.ext.webapp.util import run_wsgi_app
58 from google.appengine.ext import db
59 from models.game import Game
60 from models.game_instance import GameInstance
61 from models.message import Message
62 from server_commands import command_dict

```

```

63
64 #####
65 # Module Constants #
66 #####
67 # Operation Response Keys
68 REQUEST_TYPE_KEY = 'request_type'
69 ERROR_KEY = 'e'
70 RESPONSE_KEY = 'response'
71 GAME_ID_KEY = 'gid'
72 INSTANCE_ID_KEY = 'iid'
73 PLAYERS_KEY = 'players'
74 LEADER_KEY = 'leader'
75
76 # Request Parameter Keys
77 PLAYER_ID_KEY = 'pid'
78 INVITEE_KEY = 'inv'
79 TYPE_KEY = 'type'
80 CONTENTS_KEY = 'contents'
81 COMMAND_KEY = 'command'
82 ARGS_KEY = 'args'
83 MESSAGE_COUNT_KEY = 'count'
84 MESSAGE_RECIPIENTS_KEY = 'mrec'
85 MESSAGE_TIME_KEY = 'mtime'
86 INSTANCE_PUBLIC_KEY = 'makepublic'
87
88 #####
89 # Response Helpers #
90 #####
91
92 def run_with_response_as_transaction(req_handler, operation, *args, **
    kwargs):
93     """ Run operation in a transaction and write its response to
        req_handler.
94
95     Args:
96         req_handler: The request handler to write a response to.
97         operation: The callable function to run as a transaction.
98         args: Positional arguments to pass to operation.
99         kwargs: Keyword arguments to pass to operation.
100
101     Runs operation as a database transaction, creates an
102     OperationResponse with the return value and writes it to the
103     request handler.
104
105     If an exception raises to this function a traceback is written to
106     the debug log and an OperationResponse is written to the request
107     handler with the error message as its contents and the error key
108     set to True.
109     """
110     try:
111         response = db.run_in_transaction(operation, *args, **kwargs)
112         OperationResponse(response = response).write_to_handler(
            req_handler)
113     except BaseException, e:

```

```

114     logging.debug('exception encountered: %s' % traceback.format_exc()
115 )
116     OperationResponse(response = e.__str__(),
117                        error = True).write_to_handler(req_handler)
118
119 class OperationResponse():
120     """
121     Class for handling server operation responses and writing output
122     .
123
124     An OperationResponse is a standard way to provide a response to a
125     server request. When operations are specific to a game instance,
126     the operation response includes information about the current state
127     of that instance.
128
129     If an error is encountered during an operation the
130     OperationResponse includes only the error boolean and the error's
131     message as its response.
132
133     Attributes:
134         error: A boolean indicating that an error occurred during the
135         execution of this operation.
136         gid: The game id of the game for this operation.
137         iid: The instance id of the game instance.
138         leader: The current leader of the game instance.
139         players: A list of players in the game instance
140     """
141     def __init__(self, response, error=False):
142         """ Fill in parameters based on the error value and the model
143         returned.
144
145         Args:
146             response: If no error occurs, response should be a tuple of the
147             database model that this operation was performed with and a
148             dictionary representing the response value of the operation.
149             If an error is encountered, response should be an error
150             message.
151             error: A boolean indicating whether the operation encountered
152             an error during execution.
153
154         The OperationResponse's attributes are automatically filled in by
155         reading the attributes of the model in the response tuple. If the
156         model is a Game object then iid, leader and players are left with
157         empty values.
158         """
159         self.error = error
160         self.iid = ''
161         self.leader = ''
162         self.gid = ''
163         self.players = []
164
165         if self.error:
166             self.response = response
167         else:
168             model, self.response = response

```

```

165         if model and model.__class__.__name__ == 'GameInstance':
166             self.gid = model.parent().key().name()
167             self.iid = model.key().name()
168             self.leader = model.leader
169             self.players = model.players
170         elif model and model.__class__.__name__ == 'Game':
171             self.gid = model.key().name()
172
173     def write_to_handler(self, req_handler):
174         """ Writes a response to the req_handler.
175
176         Args:
177             req_handler: The request handler for this server request.
178
179         If the 'fmt' field of this request is 'html' then the response is
180         formatted to be written to the web. Otherwise, it is formatted to
181         be sent as json.
182         """
183         if req_handler.request.get('fmt') == 'html':
184             self.write_response_to_web(req_handler)
185         else:
186             self.write_response_to_phone(req_handler)
187
188     def write_response_to_web(self, req_handler):
189         """ Writes the response object to the request handler as html.
190
191         Args:
192             req_handler: The request handler for this server request.
193
194         Writes a web page displaying the response object as it would
195         be written to json.
196         """
197         req_handler.response.headers['Content-Type'] = 'text/html'
198         req_handler.response.out.write('<html><body>')
199         req_handler.response.out.write(''
200             <em>The server will send this to the component:</em>
201             <p />'')
202         req_handler.response.out.write(
203             self.get_response_object(req_handler.request.path))
204         req_handler.response.out.write(''
205             <p><a href="/">
206             <i>Return to Game Server Main Page</i>
207             </a>'')
208         req_handler.response.out.write('</body></html>')
209
210     def write_response_to_phone(self, req_handler):
211         """ Writes the response object to the request handler as json.
212
213         Args:
214             req_handler: The request handler for this server request.
215         """
216         req_handler.response.headers['Content-Type'] = 'application/json'
217         req_handler.response.out.write(
218             self.get_response_object(req_handler.request.path))

```

```

219
220 def get_response_object(self, request_type):
221     """ Return a JSON object as a string with the fields of this
        response.
222
223     Args:
224         request_type: The type of server request that caused this
225             operation.
226
227     Creates a dictionary out of the fields of this object and encodes
228     them in JSON.
229     """
230     response = simplejson.dumps({REQUEST_TYPE_KEY : request_type,
231                                  ERROR_KEY : self.error,
232                                  RESPONSE_KEY : self.response,
233                                  GAME_ID_KEY : self.gid,
234                                  INSTANCE_ID_KEY : self.iid,
235                                  LEADER_KEY : self.leader,
236                                  PLAYERS_KEY : self.players})
237     logging.debug('response object: %s' % response)
238     return response
239
240 #####
241 # Operation Functions #
242 #####
243
244 def get_instance_lists(gid, iid, pid):
245     """ Return the instances that a player has been invited to and
        joined.
246
247     Args:
248         gid: The game id of the Game object that this method targets.
249         iid: The instance id of the Game Instance object that this
250             method targets.
251         pid: A string containing the requesting player's email address.
252
253     The gid and pid must be valid, but the iid can be blank. This is
254     because a player must be able to query for lists of instances
255     without being in one.
256
257     Returns:
258         A tuple containing a database model and a dictionary of instance
259         lists. The database model will be a Game Instance if the gid and
260         iid parameters specify a valid GameInstance, otherwise the model
261         will be a Game. Instance lists are returned in the same format
262         as get_instance_lists_dictionary.
263
264     Raises:
265         ValueError if the game id or player id are invalid.
266     """
267     utils.check_gameid(gid)
268     player = utils.check_playerid(pid)
269     model = game = utils.get_game_model(gid)
270     if game is None:

```

```

271     game = Game(key_name = gid, instance_count = 0)
272     game.put()
273     model = game
274 elif iid:
275     instance = utils.get_instance_model(gid,iid)
276     if instance:
277         model = instance
278 instance_lists = get_instances_lists_as_dictionary(game, player)
279 return model, instance_lists
280
281 def invite_player(gid, iid, invitee):
282     """ Add invitee to the list of players invited to the specified
283         instance.
284
285     Args:
286         gid: The game id of the Game object that this method targets.
287         iid: The instance id of the Game Instance object that this
288             method targets.
289         invitee: The player id of the person to invite.
290
291     Only modifies the instance if the player has not already been
292     invited and has not joined the game.
293
294     Returns:
295         A tuple of the game instance and a single item dictionary:
296         inv: The email address of the invited player if they are
297             invited. If the player is not invited (because they have
298             already been invited or have already joined the game), the
299             value of 'inv' is the empty string.
300
301     Raises:
302         ValueError if the game id, iid or invitee email address are
303         invalid.
304     """
305     utils.check_gameid(gid)
306     utils.check_instanceid(iid)
307     player = utils.check_playerid(invitee)
308     instance = utils.get_instance_model(gid, iid)
309     if player not in instance.invited and player not in instance.players
310     :
311         instance.invited.append(player)
312         instance.put()
313     else:
314         player = ''
315     return instance, {INVITEE_KEY : player}
316
317 def join_instance(gid, iid, pid):
318     """ Attempt to add a player to an instance.
319
320     Args:
321         gid: The game id of the Game object that this method targets.
322         iid: The instance id of the Game Instance to join.
323         pid: A string containing the requesting player's email address.

```

```

323
324 A player can join a game instance if it is not full and either the
325 instance is public or the player has been invited. If this
326 operation is invoked by a player that is not current in the
327 specified instance and they are unable to join, it will fail.
328
329 If the player is already in the game instance this will succeed
330 without modifying the instance.
331
332 If the specified game instance doesn't exist, it will be created as
333 in new_instance with the specified instance id.
334
335 If no players are in the game when this player tries to join they
336 will automatically become the leader.
337
338 Returns:
339     A tuple of the game instance and the instance list dictionary
340     for this player (see get_instance_lists_as_dictionary).
341
342 Raises:
343     ValueError if the game id, instance id or player id are invalid.
344     ValueError if the player is not already in the game and is unable
345     to join.
346 """
347 utils.check_gameid(gid)
348 utils.check_instanceid(iid)
349 player = utils.check_playerid(pid)
350 instance = utils.get_instance_model(gid, iid)
351 if instance is None:
352     return new_instance(gid, iid, pid)
353 game = instance.parent()
354 instance_lists = get_instances_lists_as_dictionary(game, player)
355 instance.add_player(player)
356 instance.put()
357 if iid in instance_lists['invited']:
358     instance_lists['invited'].remove(instance.key().name())
359 if iid not in instance_lists['joined']:
360     instance_lists['joined'].append(instance.key().name())
361 return instance, instance_lists
362
363 def leave_instance(gid, iid, pid):
364     """ Remove a player from an instance.
365
366     Args:
367         gid: The game id of the game object that this method targets.
368         iid: The instance id of the Game Instance to remove the player
369             from.
370         player: The player wishing to leave the instance.
371
372     If the player that leaves the instance is the leader, the first
373     player on the players lists becomes the leader.
374
375     If no players are left, the maximum number of players allowed in
376     this instance is set to -1 so that no one may join it in the

```



```

377     future. This means that if someone tries to create an instance in
378     the future with the same instance id, they will end up with one with
379     a number appended to it (because this GameInstance object will still
380     exist).
381
382     The decision to do this was made because it is not yet possible to
383     reliably delete all of the messages in a game instance (see
384     models/game_instance.py). Thus, if players are able to join an
385     orphaned instances, the old messages could still be available. If,
386     in the future, App Engine adds ways to reliably delete database
387     models this behavior could be changed to delete the instance
388     entirely if everyone leaves.
389
390     Returns:
391         A tuple of the game object and the instance list dictionary
392         for this player (see get_instance_lists_as_dictionary).
393
394     Raises:
395         ValueError if the player is not currently in the instance.
396     """
397     utils.check_gameid(gid)
398     utils.check_instanceid(iid)
399     instance = utils.get_instance_model(gid, iid)
400     player = instance.check_player(pid)
401     instance.players.remove(player)
402     if player == instance.leader and len(instance.players) != 0:
403         instance.leader = instance.players[0]
404     if len(instance.players) == 0:
405         instance.max_players = -1
406     game = instance.parent()
407     instance_lists = get_instances_lists_as_dictionary(game, player)
408     instance_lists['joined'].remove(instance.key().name())
409     instance.put()
410     return game, instance_lists
411
412 def get_messages(gid, iid, message_type, recipient, count, time):
413     """ Retrieve messages matching the specified parameters.
414
415     Args:
416         gid: The game id of the Game object that is a parent of the
417             desired instance.
418         iid: This instance id of the Game Instance to fetch messages
419             from.
420         message_type: A string 'key' for the message. If message_type is
421             the empty string, all message types will be returned.
422         recipient: The player id of the recipient of the messages. This
423             operation will also return messages that are sent with an empty
424             recipient field.
425         count: The maximum number of messages to retrieve.
426         time: A string representation of the earliest creation time of a
427             message to returned. Must be in ISO 8601 format to parse
428             correctly.
429
430     Uses the get_messages function of the GameInstance class to

```

```

431     retrieve messages.
432
433     Returns:
434         A tuple of the game instance and a dictionary with two items:
435         'count': The number of messages returned.
436         'messages': A list of the dictionary representations of the
437             fetched messages.
438     """
439     utils.check_gameid(gid)
440     utils.check_instanceid(iid)
441     instance = utils.get_instance_model(gid, iid)
442     recipient = instance.check_player(recipient)
443     messages = instance.get_messages(count=count,
444                                     message_type=message_type,
445                                     recipient=recipient, time=time)
446     return instance, {MESSAGE_COUNT_KEY : len(messages),
447                      'messages' : messages}
448
449 def new_instance(gid, iid_prefix, pid, make_public = False):
450     """ Create a new instance of the specified game.
451
452     Args:
453         gid: The game id of the Game parent of the new instance.
454         iid_prefix: The desired instance id. If no instance has been made
455             with this name before, then this will be the instance id of the
456             newly created instance. However, since instance ids must be
457             unique, the actual instance id will likely be iid_prefix with a
458             number suffix.
459         pid: The id of the first player and leader of the game.
460         make_public: A boolean indicating whether this instance should
461             be able to be seen and joined by anyone.
462
463     The instance id will start with iid_prefix, but could have any
464     suffix. If the parent Game object does not exist, it will
465     automatically be created.
466
467     Returns:
468         A tuple of the newly created instance and an instance lists
469         dictionary (see get_instance_lists_as_dictionary).
470
471     Raises:
472         ValueError if the gameid or player id are invalid.
473     """
474     utils.check_gameid(gid)
475     player = utils.check_playerid(pid)
476     game = Game.get_by_key_name(gid)
477     if game is None:
478         game = Game(key_name = gid, instance_count = 0)
479
480     if not iid_prefix:
481         iid_prefix = player + 'instance'
482     instance = game.get_new_instance(iid_prefix, player)
483
484     instance_lists = get_instances_lists_as_dictionary(game, player)

```

```

485     instance_lists['joined'].append(instance.key().name())
486     if make_public:
487         instance.public = True
488         instance_lists['public'].append(instance.key().name())
489     instance.put()
490     game.put()
491
492     return instance, instance_lists
493
494 def new_message(gid, iid, pid, message_type, message_recipients,
495                message_content):
496     """ Create new messages and put them in the database.
497
498     Args:
499         gid: The game id of the Game parent of the instance to create a
500             message for.
501         iid: The instance id of the GameInstance to create a message for.
502         pid: The player id of the message sender.
503         message_type: A string that acts as a key for the message.
504         message_recipients: The recipients of the message formatted in
505             JSON. This can be a single player id as a JSON string, a list
506             of player ids in a JSON array or the empty string. Messages
507             sent with the empty string as a recipient can be fetched by
508             any player.
509         message_content: The string representation of a JSON value to be
510             sent as the content of the message.
511
512     Returns:
513         A tuple of the specified game instance and a dictionary with
514             two items:
515             'count' : The number of messages created.
516             'mrec' : The list of email addresses that were sent messages.
517
518     Raises:
519         ValueError if the requesting player or any of the message
520             recipients are not members of the specified game instance.
521     """
522     utils.check_gameid(gid)
523     utils.check_instanceid(iid)
524     instance = utils.get_instance_model(gid, iid)
525     player = instance.check_player(pid)
526     recipients_list = None
527     if message_recipients != '':
528         recipients_list = simplejson.loads(message_recipients)
529         if isinstance(recipients_list, basestring):
530             recipients_list = [recipients_list]
531     if not recipients_list:
532         recipients_list = ['']
533     message_list = []
534     for recipient_entry in recipients_list:
535         if recipient_entry:
536             recipient_entry = instance.check_player(recipient_entry)
537         message = Message(parent = instance,
538                           sender = player,

```

```

539             msg_type = message_type,
540             recipient = recipient_entry,
541             content = message_content)
542     message_list.append(message)
543     db.put(message_list)
544     return instance, {MESSAGE_COUNT_KEY : len(message_list),
545                     MESSAGE_RECIPIENTS_KEY : recipients_list}
546
547 def server_command(gid, iid, pid, command, arguments):
548     """ Performs the desired server command.
549
550     Args:
551         gid: The game id of the Game model for this operation.
552         iid: The instance id of the GameInstance model for
553             this operation.
554         pid: The player id of the requesting player.
555         command: The key identifying the command to execute.
556         arguments: JSON representation of arguments to the command.
557
558     If the gid and iid specify a valid game instance model it will be
559     passed to the server command. In the case that the iid is empty or
560     refers to a game instance that doesn't exist, a game model will be
561     used. Most commands will fail if passed a game model instead of a
562     game instance, but some are indifferent to the model passed to
563     them.
564
565     Unless the dynamic property do_not_put has been set to False, this
566     will put the database model after the command has been
567     performed. This means that server commands do not need to make
568     intermediate puts of the instance model passed to them.
569
570     Returns:
571         A tuple of the model used in the server command's execution and a
572         two item dictionary:
573         'type': The requested command key.
574         'contents': A Python value of the response value of the
575             command. This varies among server commands but must always be
576             able to be encoded to JSON.
577
578     Raises:
579         ValueError if the game id or player id is invalid.
580         ValueError if the arguments json cannot be parsed.
581         ValueError if command is not a known server command.
582     """
583     utils.check_gameid(gid)
584     player = utils.check_playerid(pid)
585     model = None
586     if iid:
587         model = utils.get_instance_model(gid, iid)
588     if model is None:
589         model = utils.get_game_model(gid)
590     if model is None:
591         model = Game(key_name = gid, instance_count = 0)
592

```

```

593 arguments = simplejson.loads(arguments)
594 reply = ''
595
596 if command in command_dict:
597     reply = command_dict[command](model, player, arguments)
598     if 'do_not_put' not in model.dynamic_properties() or not model.
        do_not_put:
599         model.put()
600 else:
601     raise ValueError("Invalid server command: %s." % command)
602
603 if not isinstance(reply, list):
604     reply = [reply]
605 return model, {TYPE_KEY : command, CONTENTS_KEY: reply}
606
607 def set_leader(gid, iid, pid, leader):
608     """ Set the leader of the specified instance.
609
610     Args:
611         gid: The game id of the GameInstance object's parent Game object.
612         iid: The instance id of the GameInstance to change the leader of.
613         pid: The player id of the requesting player. This player must be
614             the current instance leader in order to change the leader value.
615         leader: The player id of the new leader.
616
617     Returns:
618         A tuple of the change game instance model and a dictionary with
619         two items:
620         'current_leader' : The leader after attempting this change.
621         'leader_change' : Whether or not this attempt to set the leader
622             succeeded.
623
624     Raises:
625         ValueError if the game id or instance id are invalid.
626         ValueError if player or leader are not in the specified game
627         instance.
628     """
629     utils.check_gameid(gid)
630     utils.check_instanceid(iid)
631     instance = utils.get_instance_model(gid, iid)
632     player = instance.check_player(pid)
633     leader = instance.check_player(leader)
634     if player != instance.leader or instance.leader == leader:
635         return instance, {'current_leader' : instance.leader,
636             'leader_changed' : False}
637     instance.leader = leader
638     instance.put()
639     return instance, {'current_leader' : leader,
640         'leader_changed' : True}
641
642 def get_instance(gid, iid):
643     """ Retrieves an instance and its dictionary.
644
645     Args:
646         gid: The game id of the desired GameInstance object's parent Game

```

```

646         object.
647         iid: The instance id of the desired GameInstance object.
648
649     Returns:
650         A tuple of the game instance object and its dictionary
651         representation.
652
653     Raises:
654         ValueError if the game id or instance id are not valid.
655     """
656     utils.check_gameid(gid)
657     utils.check_instanceid(iid)
658     instance = utils.get_instance_model(gid, iid)
659     return instance, instance.to_dictionary()
660
661     #####
662     # Writer Helpers #
663     #####
664
665     def get_instances_lists_as_dictionary(game, player):
666         """ Return a dictionary with joined and invited instance id lists
667         for player.
668
669         Args:
670             game: The Game database model that is the parent of the instances
671             to query.
672             player: The email address of the player to get instance lists for.
673
674         Returns:
675             A dictionary of lists:
676             'joined' : The list of instance ids of all all instances that
677             the player has joined and not subsequently left.
678             'invited' : The list of instance ids of all instances that the
679             player has been invited to and not yet joined.
680         """
681         return {'joined' : get_instances_joined(game, player),
682                 'invited' : get_instances_invited(game, player),
683                 'public' : get_public_instances(game)}
684
685     def get_instances_joined(game, player):
686         """ Return the instance ids of instance that player has joined.
687
688         Args:
689             game: The parent Game database model to query for instances.
690             player: The email address of the player to look for in instances.
691
692         Returns:
693             An empty list if game is None. Else, returns a list of the
694             instance
695             ids of all instances with game as their parent that have player in
696             their joined list.
697         """
698         if game is None:
699             return []

```

```

698     query = game.get_joined_instance_keys_query(player)
699     return [key.name() for key in query]
700
701 def get_instances_invited(game, player):
702     """ Return the instance ids of instances that player has been
703         invited to.
704
705     Args:
706         game: The parent Game database model to query for instances.
707         player: The email address of the player to look for in instances.
708
709     Returns:
710         An empty list if game is None. Else, returns a list of the
711         instance ids of all instances with game as their parent that have
712         player in their invited list.
713     """
714     if game is None:
715         return []
716     query = game.get_invited_instance_keys_query(player)
717     return [key.name() for key in query]
718
719 def get_public_instances(game):
720     """ Return the instance ids of public instances for the specified
721         game.
722
723     Args:
724         game: The parent Game database model to query for instances.
725
726     Returns:
727         An empty list if game is None. Else, returns a list of the
728         instance ids of all joinable public instances with game as
729         their parent.
730     """
731     if game is None:
732         return []
733     query = game.get_public_instances_query(keys_only = True)
734     return [key.name() for key in query]
735
736 #####
737 # Request Handler Classes #
738 #####
739
740 class MainPage(webapp.RequestHandler):
741     """ The request handler for the index page of the game server. """
742     def get(self):
743         """Write a simple web page for displaying server information. """
744         self.response.headers['Content-Type'] = 'text/html'
745         self.response.out.write('<html><body>')
746         self.response.out.write('<h1>Game Server for App Inventor Game '
747             ' Client Component</h1>')
748         self.write_game_list()
749         self.write_methods()
750         self.response.out.write(''
751             <p><a href="http://appengine.google.com">

```

```

750     <small><i>Go to AppEngine Administration Console</i></small>
751     </a>''')
752     self.response.out.write('</body></html>')
753
754 def write_game_list(self):
755     """ Create an HTML table showing game instance information. """
756     self.response.out.write(''')
757     <p><table border=1>
758         <tr>
759             <th>Created
760             <th>Game</th>
761             <th>Instance</th>
762             <th>Players</th>
763             <th>Invitees</th>
764             <th>Leader</th>
765             <th>Public</th>
766             <th>Max Players</th>
767             <th colspan="2">More ...</th>
768         </tr>''')
769     games = db.GqlQuery("SELECT * FROM GameInstance")
770     for game in games:
771         self.response.out.write(
772             '<tr><td>%s UTC</td>\n' % game.date.ctime())
773         self.response.out.write('<td>%s</td>' % game.parent().key().name
774             ())
775         self.response.out.write('<td>%s</td>' % game.key().name())
776         self.response.out.write('<td>')
777         for player in game.players:
778             self.response.out.write(' %s' % player)
779             self.response.out.write('</td>\n')
780             self.response.out.write('<td>')
781             for invite in game.invited:
782                 self.response.out.write(' %s' % invite)
783                 self.response.out.write('</td>\n')
784                 self.response.out.write('<td>')
785                 self.response.out.write(' %s' % game.leader)
786                 self.response.out.write('</td>\n')
787                 self.response.out.write('<td>')
788                 self.response.out.write(' %s' % game.public)
789                 self.response.out.write('</td>\n')
790                 self.response.out.write('<td>')
791                 self.response.out.write(' %s' % game.max_players)
792                 self.response.out.write('</td>\n')
793                 self.response.out.write(''')
794                 <td><form action="/getinstance" method="post"
795                     enctype=application/x-www-form-urlencoded>
796                     <input type="hidden" name="gid" value="%s">
797                     <input type="hidden" name="iid" value="%s">
798                     <input type="hidden" name="fmt" value="html">
799                     <input type="submit" value="Game state"></form></td>\n''')
800                 %
801                 (game.parent().key().name(), game.key().
802                     name())
803                 self.response.out.write('</tr>')

```



```

801     self.response.out.write('</table>')
802
803 def write_methods(self):
804     """ Write links to the available server request pages. """
805     self.response.out.write(''
806         <p />Available calls:\n
807         <ul>
808         <li><a href="/newinstance">/newinstance</a></li>
809         <li><a href="/invite">/invite</a></li>
810         <li><a href="/joininstance">/joininstance</a></li>
811         <li><a href="/leaveinstance">/leaveinstance</a></li>
812         <li><a href="/newmessage">/newmessage</a></li>
813         <li><a href="/messages">/messages</a></li>
814         <li><a href="/setleader">/setleader</a></li>
815         <li><a href="/getinstance">/getinstance</a></li>
816         <li><a href="/getinstancelists">/getinstancelists</a></li>
817         <li><a href="/servercommand">/servercommand</a></li>
818         </ul>''')
819
820 class GetInstanceLists(webapp.RequestHandler):
821     """ Request handler for the get_instance_lists operation. """
822     def post(self):
823         """ Execute get_instance_lists and write the response to the
824             handler.
825
826             Request parameters:
827             gid: The game id of the parent Game to get instances of.
828             iid: The instance id of the game instance to execute the
829                 command with. This is optional for this command, although,
830                 including it will result in the ResponseObject including
831                 leader and player information.
832             pid: The player id of the requesting player.
833         """
834         logging.debug('/getinstancelists?%s\n%s|' %
835             (self.request.query_string, self.request.body))
836         gid = self.request.get(GAME_ID_KEY)
837         iid = self.request.get(INSTANCE_ID_KEY)
838         pid = self.request.get(PLAYER_ID_KEY)
839         run_with_response_as_transaction(self, get_instance_lists, gid,
840             iid, pid)
841
842     def get(self):
843         """ Write a short HTML form to perform a get_instance_lists
844             operation. """
845         self.response.out.write(''
846             <html><body>
847             <form action="/getinstancelists" method="post"
848                 enctype=application/x-www-form-urlencoded>
849                 <p>Game ID <input type="text" name="gid" /></p>
850                 <p>Instance ID <input type="text" name="iid" /></p>
851                 <p>Player ID <input type="text" name="pid" /></p>
852                 <input type="hidden" name="fmt" value="html">
853                 <input type="submit" value="Get Instance Lists">
854             </form>''')

```

```

852     self.response.out.write('</body></html>\n')
853
854 class GetMessages(webapp.RequestHandler):
855     """ Request handler for the get_messages operation. """
856     def post(self):
857         """ Execute get_messages and write the response to the handler.
858
859         Request parameters:
860             gid: The game id of the parent Game.
861             iid: The instance id of the game instance to execute the
862                 command with.
863             pid: The player id of the message recipient.
864             type: The type of messages requested or the empty string to
865                 retrieve all messages.
866             count: An integer number of messages to retrieve. This is
867                 treated as a maximum and defaults to 1000 if there is a
868                 failure retrieving the count parameter.
869             mtime: A string in ISO 8601 date format. All messages returned
870                 will have a creation time later than this time. Defaults to
871                 datetime.min if there is a failure in retrieving or parsing
872                 the parameter.
873         """
874         logging.debug('/messages?%s\n%s|' %
875                       (self.request.query_string, self.request.body))
876         gid = self.request.get(GAME_ID_KEY)
877         iid = self.request.get(INSTANCE_ID_KEY)
878         message_type = self.request.get(TYPE_KEY)
879         recipient = self.request.get(PLAYER_ID_KEY)
880
881         count = 1000
882         try:
883             count = int(self.request.get(MESSAGE_COUNT_KEY))
884         except ValueError:
885             pass
886
887         time = datetime.min
888         try:
889             time_string = self.request.get(MESSAGE_TIME_KEY)
890             if time_string is not None and time_string != '':
891                 time = iso8601.parse_date(time_string)
892         except ValueError:
893             pass
894
895         run_with_response_as_transaction(self, get_messages, gid, iid,
896                                         message_type, recipient, count,
897                                         time)
898     def get(self):
899         """ Write a short HTML form to perform a get_messages operation.
900
901         """
902         self.response.out.write('''
903         <html><body>
904         <form action="/messages" method="post"
905             enctype=application/x-www-form-urlencoded>

```

```

904         <p>Game ID <input type="text" name="gid" /></p>
905         <p>Instance ID <input type="text" name="iid" /></p>
906         <p>Message type <input type="text" name="type" /> </p>
907         <p>Email <input type="text" name="pid" /> </p>
908         <p>Count <input type="text" name="count" /> </p>
909         <p>Time <input type="text" name="mtime" /> </p>
910         <input type="hidden" name="fmt" value="html">
911         <input type="submit" value="Get Messages">
912     </form></body></html>\n''')
913
914     class InvitePlayer(webapp.RequestHandler):
915         """ Request handler for the invite_player operation."""
916         def post(self):
917             """ Execute invite_player and write the response to the handler.
918
919             Request parameters:
920                 gid: The game id of the parent Game.
921                 iid: The instance id of the game instance to invite the
922                     player to.
923                 pid: The player id of the requesting player.
924                 inv: The player id of the player to invite.
925             """
926             logging.debug('/invite?%s\n|%s|' %
927                           (self.request.query_string, self.request.body))
928             gid = self.request.get(GAME_ID_KEY)
929             iid = self.request.get(INSTANCE_ID_KEY)
930             inv = self.request.get(INVITEE_KEY)
931             run_with_response_as_transaction(self, invite_player, gid, iid,
932                                             inv)
933
934         def get(self):
935             """ Write a short HTML form to perform an invite_player operation.
936             """
937             self.response.out.write('''
938             <html><body>
939             <form action="/invite" method="post"
940                 enctype=application/x-www-form-urlencoded>
941                 <p>Game ID <input type="text" name="gid" /></p>
942                 <p>Instance ID <input type="text" name="iid" /></p>
943                 <p>Player ID <input type="text" name="pid" /></p>
944                 <p>Invitee <input type="text" name="inv" /> </p>
945                 <input type="hidden" name="fmt" value="html">
946                 <input type="submit" value="Invite player">
947             </form>''')
948             self.response.out.write('</body></html>\n')
949
950     class JoinInstance(webapp.RequestHandler):
951         """ Request handler for the join_instance operation."""
952         def post(self):
953             """ Execute join_instance and write the response to the handler.
954
955             Request parameters:
956                 gid: The game id of the parent Game.
957                 iid: The instance id of the game instance to join.

```

```

956         pid: The player id of the requesting player.
957         """
958         logging.debug('/joininstance?%s\n|%s|' %
959                       (self.request.query_string, self.request.body))
960         gid = self.request.get(GAME_ID_KEY)
961         iid = self.request.get(INSTANCE_ID_KEY)
962         pid = self.request.get(PLAYER_ID_KEY)
963         run_with_response_as_transaction(self, join_instance, gid, iid,
964                                         pid)
965
966     def get(self):
967         """ Write a short HTML form to perform a join_instance operation.
968         """
969         self.response.out.write('''
970 <html><body>
971 <form action="/joininstance" method="post"
972       enctype=application/x-www-form-urlencoded>
973   <p>Game ID <input type="text" name="gid" /></p>
974   <p>Instance ID <input type="text" name="iid" /></p>
975   <p>Player ID <input type="text" name="pid" /> </p>
976   <input type="hidden" name="fmt" value="html">
977   <input type="submit" value="Join Instance">
978 </form>''')
979         self.response.out.write('</body></html>\n')
980
981     class LeaveInstance(webapp.RequestHandler):
982         """ Request handler for the leave_instance operation. """
983         def post(self):
984             """ Execute leave_instance and write the response to the handler.
985
986             Request parameters:
987             gid: The game id of the parent Game.
988             iid: The instance id of the game instance to leave.
989             pid: The player id of the requesting player.
990             """
991             logging.debug('/leaveinstance?%s\n|%s|' %
992                           (self.request.query_string, self.request.body))
993             gid = self.request.get(GAME_ID_KEY)
994             iid = self.request.get(INSTANCE_ID_KEY)
995             pid = self.request.get(PLAYER_ID_KEY)
996             run_with_response_as_transaction(self, leave_instance, gid, iid,
997                                             pid)
998
999     def get(self):
1000         """ Write a short HTML form to perform a leave_instance operation.
1001         """
1002         self.response.out.write('''
1003 <html><body>
1004 <form action="/leaveinstance" method="post"
1005       enctype=application/x-www-form-urlencoded>
1006   <p>Game ID <input type="text" name="gid" /></p>
1007   <p>Instance ID <input type="text" name="iid" /></p>
1008   <p>Player ID <input type="text" name="pid" /> </p>
1009   <input type="hidden" name="fmt" value="html">

```

```

1006         <input type="submit" value="Leave Instance">
1007     </form>''')
1008     self.response.out.write('</body></html>\n')
1009
1010
1011 class NewInstance(webapp.RequestHandler):
1012     """ Request handler for the new_instance operation. """
1013     def post(self):
1014         """ Execute new_instance and write the response to the handler.
1015
1016         Request parameters:
1017             gid: The game id of the parent Game.
1018             iid: The proposed instance id of the new instance. The instance
1019                 id of the created instance could differ from this if the
1020                 proposed id is already in use.
1021             pid: The player id of the requesting player.
1022             make_public: A boolean indicating whether this instance should
1023                 be able to be seen and joined by anyone.
1024         """
1025         logging.debug('/newinstance?%s\n|%s|' %
1026                       (self.request.query_string, self.request.body))
1027         gid = self.request.get(GAME_ID_KEY)
1028         iid = self.request.get(INSTANCE_ID_KEY)
1029         pid = self.request.get(PLAYER_ID_KEY)
1030         make_public = False
1031         try:
1032             make_public = utils.get_boolean(self.request.get(
1033                 INSTANCE_PUBLIC_KEY))
1034         except ValueError:
1035             pass
1036         run_with_response_as_transaction(self, new_instance, gid, iid,
1037                                         pid, make_public)
1038
1039     def get(self):
1040         """ Write a short HTML form to perform a new_instance operation.
1041         """
1042         self.response.out.write('''
1043         <html><body>
1044         <form action="/newinstance" method="post"
1045             enctype=application/x-www-form-urlencoded>
1046             <p>Game ID <input type="text" name="gid" /></p>
1047             <p>Instance ID <input type="text" name="iid" /></p>
1048             <p>First player ID <input type="text" name="pid" /></p>
1049             <input type="hidden" name="fmt" value="html">
1050             <input type="submit" value="New Instance">
1051         </form></body></html>\n''')
1052
1053 class NewMessage(webapp.RequestHandler):
1054     """ Request handler for the new_message operation. """
1055     def post(self):
1056         """ Execute new_message and write the response to the handler.
1057
1058         Request parameters:
1059             gid: The game id of the parent Game.

```

```

1058         iid: The instance id of the game instance add messages to.
1059         pid: The player id of the requesting player.
1060         type: The message type key.
1061         mrec: Json representation of the recipients of the message.
1062         content: Json representation of the contents of the message.
1063     """
1064     logging.debug('/newmessage?%s\n|%s|' %
1065                  (self.request.query_string, self.request.body))
1066     gid = self.request.get(GAME_ID_KEY)
1067     pid = self.request.get(PLAYER_ID_KEY)
1068     iid = self.request.get(INSTANCE_ID_KEY)
1069     message_type = self.request.get(TYPE_KEY)
1070     message_recipients = self.request.get(MESSAGE_RECIPIENTS_KEY)
1071     message_content = self.request.get(CONTENTS_KEY)
1072     run_with_response_as_transaction(self, new_message, gid, iid, pid,
1073                                     message_type, message_recipients,
1074                                     message_content)
1075
1076     def get(self):
1077         """ Write a short HTML form to perform a new_message operation. """
1078         self.response.out.write(''
1079 <html><body>
1080 <form action="/newmessage" method="post"
1081     enctype=application/x-www-form-urlencoded>
1082     <p>Game ID <input type="text" name="gid" /></p>
1083     <p>Instance ID <input type="text" name="iid" /></p>
1084     <p>Player ID <input type="text" name="pid" /></p>
1085     <p>Message type <input type="text" name="type" /> </p>
1086     <p>Message Recipients (Json array) <input type="text" name="
mrec" />
1087     </p>
1088     <p>Message Contents (Json array) <input type="text" name="
contents" />
1089     </p>
1090     <input type="hidden" name="fmt" value="html">
1091     <input type="submit" value="Send Message">
1092 </form>''')
1093     self.response.out.write(''<p> Expected format for recipients: <br>
>
1094         ["email@domain.com", "email2@domain.com"]
</p>''')
1095     self.response.out.write(''<p> Expected format for contents: <br>
1096         ["string 1", "string 2"] </p>''')
1097     self.response.out.write(''</body></html>\n'')
1098
1099
1100     class ServerCommand(webapp.RequestHandler):
1101         """ Request handler for the server_command operation. """
1102         def post(self):
1103             """ Execute server_command and write the response to the handler.
1104
1105             Request parameters:
1106             gid: The game id of the parent Game to execute the command
1107             with.

```

```

1108         iid: The instance id of the game instance to execute the
1109             command with.
1110         pid: The player id of the requesting player.
1111         command: The key of the command.
1112         arguments: Json representation of the arguments to the
1113             server command.
1114     """
1115     logging.debug('/servercommand?%s\n|%s|' %
1116                  (self.request.query_string, self.request.body))
1117     gid = self.request.get(GAME_ID_KEY)
1118     iid = self.request.get(INSTANCE_ID_KEY)
1119     pid = self.request.get(PLAYER_ID_KEY)
1120     command = self.request.get(COMMAND_KEY)
1121     arguments = self.request.get(ARGS_KEY)
1122     run_with_response_as_transaction(self, server_command, gid, iid,
1123                                     pid,
1124                                     command, arguments)
1125
1126     def get(self):
1127         """ Write a short HTML form to perform a set_leader operation. """
1128         self.response.out.write(''
1129 <html><body>
1130 <form action="/servercommand" method="post"
1131     enctype=application/x-www-form-urlencoded>
1132     <p>Game ID <input type="text" name="gid" /></p>
1133     <p>Instance ID <input type="text" name="iid" /></p>
1134     <p>Player ID <input type="text" name="pid" /> </p>
1135     <p>Command <input type="text" name="command" /> </p>
1136     <p>Arguments (Json array) <input type="text" name="args" /> </p>
1137 >
1138     <input type="hidden" name="fmt" value="html">
1139     <input type="submit" value="Send Command">
1140 </form>''')
1141     self.response.out.write(''</body></html>\n'')
1142
1143     class SetLeader(webapp.RequestHandler):
1144         """ Request handler for the set_leader operation. """
1145         def post(self):
1146             """ Execute set_leader and write the response to the handler.
1147
1148             Request parameters:
1149             gid: The game id of the parent Game.
1150             iid: The instance id of the game instance to change the
1151                 leader of.
1152             leader: The player id of the new leader candidate.
1153             pid: The player id of the requesting player.
1154             """
1155             logging.debug('/setleader?%s\n|%s|' %
1156                          (self.request.query_string, self.request.body))
1157             gid = self.request.get(GAME_ID_KEY)
1158             iid = self.request.get(INSTANCE_ID_KEY)
1159             leader = self.request.get(LEADER_KEY)
1160             pid = self.request.get(PLAYER_ID_KEY)

```

```

1160     run_with_response_as_transaction(self, set_leader, gid, iid, pid,
1161                                     leader)
1162
1163 def get(self):
1164     """ Write a short HTML form to perform a set_leader operation."""
1165     self.response.out.write(''
1166                             <html><body>
1167                             <form action="/setleader" method="post"
1168                                 enctype=application/x-www-form-urlencoded>
1169                                 <p>Game ID <input type="text" name="gid" /></p>
1170                                 <p>Instance ID <input type="text" name="iid" /></p>
1171                                 <p>Player ID <input type="text" name="pid" /></p>
1172                                 <p>New leader (player id) <input type="text" name="leader" />
1173                                 </p>
1174                                 <input type="hidden" name="fmt" value="html">
1175                                 <input type="submit" value="Set leader">
1176                                 </form>''')
1177     self.response.out.write('</body></html>\n')
1178
1179 #####
1180 # Handlers not used by GameClient component #
1181 #####
1182
1183 class GetInstance(webapp.RequestHandler):
1184     """ Request handler for the get_instance operation."""
1185     def post(self):
1186         """ Execute get_instance and write the response to the handler.
1187
1188         Request parameters:
1189         gid: The game id of the parent Game.
1190         iid: The instance id of the game instance to get the
1191             information of.
1192         """
1193         logging.debug('/getinstance?%s\n|%s|' %
1194                       (self.request.query_string, self.request.body))
1195         gid = self.request.get(GAME_ID_KEY)
1196         iid = self.request.get(INSTANCE_ID_KEY)
1197         run_with_response_as_transaction(self, get_instance, gid, iid)
1198
1199 def get(self):
1200     """ Write a short HTML form to perform a get_instance operation.
1201     """
1202     self.response.out.write(''
1203                             <html><body>
1204                             <form action="/getinstance" method="post"
1205                                 enctype=application/x-www-form-urlencoded>
1206                                 <p>Game ID <input type="text" name="gid" /></p>
1207                                 <p>Instance ID <input type="text" name="iid" /></p>
1208                                 <input type="hidden" name="fmt" value="html">
1209                                 <input type="submit" value="Get Instance Info">
1210                                 </form>''')
1211     self.response.out.write('</body></html>\n')
1212
1213 #####

```



```

1211 # Application definition #
1212 #####
1213
1214 def application(custom_command_dict):
1215     """ Return the WSGI Application with the game server request
        handlers.
1216
1217     Args:
1218         custom_command_dict: A dictionary of command name strings
1219                             to functions.
1220
1221     The custom_command_dict will be added to the server's command
1222     dictionary so that custom commands can be invoked with the
1223     ServerCommand request handler. If command names in
1224     custom_command_dict are the same as built in server commands they
1225     will overwrite the built in functions.
1226     """
1227     for command in custom_command_dict.iteritems():
1228         command_dict[command[0]] = command[1]
1229     return webapp.WSGIApplication([('/', MainPage),
1230                                   ('/newinstance', NewInstance),
1231                                   ('/invite', InvitePlayer),
1232                                   ('/joininstance', JoinInstance),
1233                                   ('/leaveinstance', LeaveInstance),
1234                                   ('/newmessage', NewMessage),
1235                                   ('/getinstancelists',
1236                                    GetInstanceLists),
1237                                   ('/messages', GetMessages),
1238                                   ('/setleader', SetLeader),
1239                                   ('/servercommand', ServerCommand),
1240                                   ('/getinstance', GetInstance)],
                                   debug=True)

```

A.2: server_commands.py - Contains the server command dictionary and built in server commands relating to instance management.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 Contains the server commands dictionary as well as the implementation
15 of a number of server commands related to instance management. More

```

```

16 specific server commands are in the extensions folder.
17
18 To enable server commands, they must be entered into
19 commands_dict. Every server command should take in a database model
20 (either a game instance or a game), the email address of the player
21 that requested the server command and a list of arguments. The format
22 of the arguments and what is done with the player and the database
23 model depends on the command.
24
25 Server commands should be generally useful to creators of games and
26 not be created for a specific game. Additionally, command functions
27 should, where appropriate, only parse the arguments and perform their
28 actual operations in separate methods. This allows for custom modules
29 to utilize extensions more easily when creating game specific
30 functions.
31 """
32
33 __authors__ = ["Bill Magnuson" <billmag@mit.edu>']
34
35 import logging
36 import traceback
37 import utils
38 from google.appengine.api import mail
39 from google.appengine.ext import db
40 from google.appengine.runtime import apiproxy_errors
41 from django.utils import simplejson
42 from extensions import scoreboard
43 from extensions import card_game
44
45 EMAIL_SENDER = ""
46
47 def send_email_command(model, player, arguments):
48     """ Send an email using App Engine's email server.
49
50     Args:
51         instance: Not used, can be any value.
52         player: The player requesting that the email be sent.
53         arguments: A two item list with the subject of the email as the
54             first item and the body of the email as the second.
55
56     EMAIL_SENDER must be defined above and be a listed developer
57     of your AppEngine application for this to work successfully.
58
59     Returns:
60         True if the email sends successfully, False otherwise.
61     """
62     if email_sender:
63         message_recipient = arguments[0]
64         message_content = arguments[1]
65         mail.send_mail(sender=EMAIL_SENDER,
66                       to=message_recipient,
67                       subject=message_content[0],
68                       body=message_content[1] +
69                       '\nMessage sent from AppInventorGameServer by ' +

```

```

70         player + '.')
```

```

71     return True
72     return False
73
74 def set_public_command(instance, player, arguments):
75     """ Set the public membership field for an instance.
76
77     Args:
78         instance: The GameInstance database model to change.
79         player: The player requesting the change. This player must
80             be the current leader of the instance.
81         arguments: A single item list containing the desired
82             boolean value for the public field of instance.
83
84     A public game can be joined by players without first being
85     invited. Changing the value of public does not change the current
86     membership of the game.
87
88     Returns:
89         The new value of public for the instance.
90
91     Raises:
92         ValueError if the requesting player is not the leader of the
93             instance.
94         ValueError if the argument is unable to be parsed into a boolean.
95     """
96     instance.check_leader(player)
97     value = utils.get_boolean(arguments[0])
98     instance.public = value
99     return value
100
101 def set_max_players_command(instance, player, arguments):
102     """ Set the maximum number of players allowed to join an instance.
103
104     Args:
105         instance: The GameInstance database model to change.
106         player: The player requesting the change. This player must be the
107             current leader of the instance.
108         arguments: A single item list containing the desired integer value
109             for the max players of this instance.
110
111     If the maximum player count is set to a value lower than the current
112     number of players, no players will be removed. However, new players
113     will not be able to join until the max players count goes up or
114     enough players leave the instance that the number of players is less
115     than the maximum.
116
117     Returns:
118         The new value of max_players for the instance.
119
120     Raises:
121         ValueError if the requesting player is not the leader of the
122             instance.
123         ValueError if the argument is unable to be parsed into an integer.
```

```

124     """
125     instance.check_leader(player)
126     max_players = int(arguments[0])
127     instance.max_players = max_players
128     return max_players
129
130 def get_public_instances_command(model, player, arguments = None):
131     """ Return a list of public instances of the specified game.
132
133     Args:
134         model: Either a Game or GameInstance database model. If model is a
135             GameInstance, this will return the public instances of its
136             parent Game.
137         player: Not used. Value can be anything.
138         arguments: Not used, can be any value.
139
140     Returns:
141         A list of all public instances in this game that have less players
142         than their maximum (i.e. can be joined). Instances are sorted with
143         the newest ones first. Each entry in the list of instances is
144         itself a three item list with the instance id as the first item,
145         the number of players currently in the game as the second item and
146         the maximum number of players (if any) as the third item. If no
147         maximum number of players is set for the game instance the third
148         item will be set to zero.
149     """
150     game = utils.get_game(model)
151     public_instances = game.get_public_instances_query().fetch(1000)
152     return [[i.key().name(), len(i.players), i.max_players]
153             for i in public_instances]
154
155 def delete_instance_command(instance, player, arguments = None):
156     """ Delete an instance and its messages.
157
158     Args:
159         instance: The instance to delete.
160         player: The player requesting the deletion. This player must
161             be the current leader of the instance.
162         arguments: Not used, can be any value.
163
164     Makes a good faith effort to delete the messages, but deleting large
165     numbers of database entries is currently very buggy in
166     AppEngine. This will hopefully get better over time as AppEngine
167     advances. See the method delete_messages in models/game_instance.py
168     for more information.
169
170     If the deletion of messages fails the exception will be logged and
171     this command will return normally.
172
173     Returns:
174         True if the instance deletes successfully.
175
176     Raises:
177         ValueError if player is not the leader of the instance.

```

```

178     ValueError if instance is not a GameInstance model.
179     """
180     if instance.__class__.__name__ != 'GameInstance':
181         raise ValueError("Only models of type GameInstance may be deleted.
182                             ")
183
184     instance.check_leader(player)
185     try:
186         instance.delete_messages()
187     except apiproxy_errors.ApplicationError, err:
188         logging.debug("Exception during message deletion: %s" %
189                       traceback.format_exc())
190     db.delete(instance)
191     instance.do_not_put = True
192     return True
193
194 def decline_invite_command(instance, player, arguments = None):
195     """ Remove a player from the invited list of an instance.
196
197     Args:
198         instance: The instance to uninvite player from.
199         player: The player wishing to decline an invite.
200         arguments: Not used, can be any value.
201
202     If the player wasn't actually invited to the game, nothing happens
203     and this method returns false.
204
205     Returns:
206         True if the player was previously invited to the game, False
207         otherwise.
208     """
209     if player in instance.invited:
210         instance.invited.remove(player)
211         return True
212     return False
213
214 command_dict = {
215     'sys_email' : send_email_command,
216     'sys_set_public' : set_public_command,
217     'sys_set_max_players' : set_max_players_command,
218     'sys_get_public_instances' : get_public_instances_command,
219     'sys_delete_instance' : delete_instance_command,
220     'sys_decline_invite' : decline_invite_command,
221
222     # Scoreboard commands.
223     'scb_get_scoreboard' : scoreboard.get_scoreboard_command,
224     'scb_get_score' : scoreboard.get_score_command,
225     'scb_add_to_score' : scoreboard.add_to_score_command,
226     'scb_set_score' : scoreboard.set_score_command,
227     'scb_clear_scoreboard' : scoreboard.clear_scoreboard_command,
228
229     # Card commands.
230     'crd_set_deck' : card_game.set_deck_command,
231     'crd_deal_cards' : card_game.deal_cards_command,

```

```

231     'crd_draw_cards' : card_game.draw_cards_command,
232     'crd_discard' : card_game.discard_command,
233     'crd_pass_cards' : card_game.pass_cards_to_player_command,
234     'crd_cards_left' : card_game.get_cards_remaining_command
235 }

```

A.3: utils.py - Helper utility functions for input sanitizing and database operations.

```

1  # Copyright 2010 Google Inc.
2
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6
7  #     http://www.apache.org/licenses/LICENSE-2.0
8
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 """
15 Utility functions for input validation and database model access.
16 """
17
18 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
19
20 import re
21 from google.appengine.ext import db
22 from google.appengine.ext.db import Key
23
24 EMAIL_ADDRESS_REGEX = ("([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@"
25                        "([-0-9a-zA-Z]+[.])+[a-zA-Z]{2,6}")
26
27 def get_game_model(gid):
28     """ Return a Game model for the given game id.
29
30     Args:
31         gid: The game id of the Game.
32
33     Returns:
34         The database model for the specified id or None if no such model
35         exists.
36     """
37     game_key = Key.from_path('Game', gid)
38     model = db.get(game_key)
39     return model
40
41 def get_instance_model(gid, iid):
42     """ Return a GameInstance model for the given game and instance ids.
43
44     Args:

```

```

45     gid: The game id of the GameInstance.
46     iid: The instance id of the GameInstance.
47
48     Returns:
49         The database model for the specified ids or None if the
50         GameInstance doesn't exist..
51     """
52     instance_key = Key.from_path('Game', gid, 'GameInstance', iid)
53     model = db.get(instance_key)
54     return model
55
56 def check_playerid(pid, instance = None):
57     """ Return a valid player id.
58
59     Args:
60         pid: A string containing the email address of the player or the
61             special identified 'leader'.
62         instance: (optional) The instance from which to fetch the leader
63             from when pid is 'leader'.
64
65     Returns:
66         Strips the supplied player id of superfluous characters and
67         returns only the email address. Also does conversion of the
68         special string 'leader' to the current leader of instance.
69
70     Raises:
71         ValueError if pid does not match an email address regular
72         expression.
73     """
74     if instance and pid.lower() == 'leader':
75         pid = instance.leader
76
77     if pid is None or pid == "":
78         raise ValueError('The player identifier is blank.')
79     stripped_email = re.search(EMAIL_ADDRESS_REGEX, pid)
80     if stripped_email is None:
81         raise ValueError('%s is not a valid email address.' % pid)
82     return stripped_email.group(0)
83
84 def check_gameid(gid):
85     """ Validate the game id to make sure it is not empty.
86
87     Args:
88         gid: The game id to check
89
90     Returns:
91         The game id.
92
93     Raises:
94         ValueError if the game id is the empty string or None.
95     """
96     if gid == "" or gid is None:
97         raise ValueError('Bad Game Id: %s' % gid)
98     return gid

```

```

99
100 def check_instanceid(iid):
101     """ Validate the instance id to make sure it is not empty.
102
103     Args:
104         iid: The instance id to check
105
106     Returns:
107         The instance id.
108
109     Raises:
110         ValueError if the instance id is the empty string or None.
111     """
112     if iid == "" or iid is None:
113         raise ValueError('No instance specified for request.')
114     return iid
115
116 def get_boolean(value):
117     """ Return a bool from value.
118
119     Args:
120         value: A string or bool representing a boolean.
121
122     Returns:
123         If value is a bool, value is returned without modification.
124
125         If value is a string this will convert 'true' and 'false'
126         (ignoring case) to their associated values.
127
128     Raises:
129         ValueError if value does not match one of the string tests and is
130         not a bool.
131     """
132     if type(value) is not bool:
133         value = value.lower()
134         if value == 'true':
135             value = True
136         elif value == 'false':
137             value = False
138         else:
139             raise ValueError("Boolean value was not valid")
140     return value
141
142 def get_game(model):
143     """ Return a Game object.
144
145     Args:
146         model: A database model that is either a GameInstance or Game.
147
148     Returns:
149         Either returns model or its parent if either of them is a Game
150         object.
151
152     Raises:

```



```

153     ValueError if either model or its parent is not a Game object.
154     """
155     if model.__class__.__name__ == 'GameInstance':
156         model = model.parent()
157     if model.__class__.__name__ == 'Game':
158         return model
159     raise ValueError('Invalid model passed to get_game')

```

A.1.1 Models

A.4: game.py - The game database model.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 The Game database model class and associated methods.
15 """
16
17 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
18
19 from google.appengine.ext import db
20 from game_instance import GameInstance
21
22 class Game(db.Model):
23     """ A model for a game type.
24
25     A Game is the parent object of GameInstance objects. Each Game
26     object should correspond to a type of game. Class methods are
27     available that provide queries to discover GameInstance objects
28     with this parent.
29
30     The key_name of a Game should be the game's name.
31
32     Attributes:
33         instance_count: The number of instances that have been made with
34             this Game as their parent. This number is managed manually.
35     """
36     instance_count = db.IntegerProperty(default=0)
37
38     def get_new_instance(self, prefix, player):
39         """ Create a new GameInstance and return its model.

```

```

40
41     Args:
42         prefix: A string used as the beginning of the instance id.
43         player: The email address of the player.
44
45     When this returns, neither this Game model or the new
46     GameInstance have been put() in the database. If the GameInstance
47     should persist, both models need to be put().
48
49     Returns:
50         A GameInstance object with a unique instance id beginning with
51         prefix and player as the leader and sole member of the
52         instance.
53     """
54     prefix = prefix.replace(' ', '')
55     new_iid = prefix
56     self.instance_count += 1
57     new_index = self.instance_count
58     while GameInstance.get_by_key_name(new_iid, parent=self) is not
59     None:
60         new_index += 1
61         new_iid = prefix + str(new_index)
62     instance = GameInstance(parent = self, key_name = new_iid,
63                             players = [player], leader = player)
64     return instance
65
66 def get_public_instances_query(self, keys_only = False):
67     """ Return a query object for public instances of this game.
68
69     Args:
70         keys_only (optional): Whether this database query should return
71         only keys, or entire models.
72
73     Returns:
74         A query object of all public game instances that are not full
75         in order of creation time from oldest to newest. Any instance
76         returned by this query should be able to be joined by any
77         player at the time the results are fetched.
78     """
79     query = GameInstance.all(keys_only = keys_only)
80     query.filter("public =", True)
81     query.filter("full =", False)
82     query.ancestor(self.key())
83     query.order('-date')
84     return query
85
86 def get_invited_instance_keys_query(self, player):
87     """ Return a query object for instances a player has been invited
88     to.
89
90     Args:
91         player: The email address of the player.
92
93     Returns:

```

```

92     A query object of all game instances that player has been
93     invited to and that are not full in order of creation time from
94     oldest to newest. Any instance returned by this query should be
95     able to be joined by the player at the time the results are
96     fetched.
97     """
98     query = GameInstance.all(keys_only = True)
99     query.filter("invited =", player)
100    query.filter("full =", False)
101    query.ancestor(self.key())
102    query.order('-date')
103    return query
104
105    def get_joined_instance_keys_query(self, player):
106        """ Return a query object for instances a player has already
107        joined.
108
109        Args:
110            player: The email address of the player.
111
112        Returns:
113            A query object of all game instances that player has joined in
114            order of creation time from oldest to newest.
115        """
116        query = GameInstance.all(keys_only = True)
117        query.filter("players =", player)
118        query.ancestor(self.key())
119        query.order('-date')
120        return query

```

A.5: game_instance.py - The game instance database model.

```

1  # Copyright 2009 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
11 # implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 """ The GameInstance database model class and associated methods."""
15 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
16
17 from datetime import datetime
18 from django.utils import simplejson
19 from game_server import utils
20 from google.appengine.ext import db

```

```

21 from message import Message
22
23 class GameInstance(db.Expando):
24     """ A model for an instance of a game.
25
26     A GameInstance contains all of the membership and message
27     information for an instance of a particular game. It is implemented
28     as an Expando model to allow extensions and custom modules to add
29     dynamic properties in order to extend the functionality of the
30     instance.
31
32     The key_name of a GameInstance should be the unique instance id. A
33     GameInstance's parent is the Game model that it is an instance of.
34
35     Attributes:
36         players: A list of the email addresses of players currently in the
37                 instance.
38         invited: A list of email addresses of invited players.
39         leader: The player that is currently the leader of the instance.
40         date: The date of creation, automatically set upon instantiation.
41         public: A bool that determines whether a player must first be
42                invited before they can join this instance.
43         full: A boolean indicating whether or not the game has reached
44              its maximum membership. Automatically set when the GameInstance
45              is put.
46         max_players: An integer for the maximum number of players allowed
47                     in this instance or 0 if there is no maximum.
48
49     """
50     players = db.StringListProperty(required=True)
51     invited = db.StringListProperty(default=[])
52     leader = db.StringProperty(required=True)
53     date = db.DateTimeProperty(required=True, auto_now=True)
54     public = db.BooleanProperty(default=False)
55     full = db.BooleanProperty(default=False)
56     max_players = db.IntegerProperty(default=0)
57
58     def put(self):
59         """ Set the value of full and put this instance in the database.
60         """
61         self.set_full()
62         db.Model.put(self)
63
64     def set_full(self):
65         """ Set the full attribute of this entity appropriately.
66
67         This should be called at put time to make sure that any stored
68         GameInstance model has the appropriate value for full. A game is
69         full if it has a non-zero value for max players which is less than
70         or equal to the number of players in the game.
71         """
72         if self.max_players == 0 or self.max_players > len(self.players):
73             self.full = False
74         else:

```

```

74         self.full = True
75
76     def to_dictionary(self):
77         """ Return a dictionary representation of the instance's
78         attributes. """
79         return {'gameid' : self.parent().key().name(),
80                 'instanceId' : self.key().name(),
81                 'leader' : self.leader,
82                 'players' : self.players,
83                 'invited' : self.invited,
84                 'public' : self.public,
85                 'max_players' : self.max_players}
86
87     def __str__(self):
88         """ Return a json string of this model's dictionary. """
89         return simplejson.dumps(self.to_dictionary())
90
91     def create_message(self, sender, msg_type, recipient, content):
92         """ Create a new message model with this instance as its parent.
93
94         Args:
95             sender: A string describing the creator of the message.
96             msg_type: A string that acts as a key for the message.
97             recipient: The intended recipient of this message. The
98                       recipient should either be the empty string or an email
99                       address. Messages sent with the empty string as their
100                      recipient can be fetched by any player.
101             content: A python list or dictionary representing the content
102                     of this message. Converted to a json string for storage.
103
104         Returns:
105             A new Message model with the specified attributes. This model
106             has not yet been put in the database.
107         """
108         return Message(parent = self, sender = sender, msg_type = msg_type
109                        ,
110                        recipient = recipient, content = simplejson.dumps(
111                        content))
112
113     def get_messages(self, time = datetime.min, count = 1000,
114                     message_type='', recipient=''):
115         """ Return a list of message dictionaries using query_messages.
116
117         Args (optional):
118             time: (default: datetime.min) All messages retrieved must have
119                   been created after this time..
120             count: (default 1000) The maximum number of messages to
121                   retrieve.
122             message_type: (default '') The message type to retrieve. If
123                           left as None or the empty string then all messages matching
124                           other criteria will be returned.
125             recipient: (default '') The recipient of the messages to
126                        retrieve. All messages sent with a recipient of the empty
127                        string will also be retrieved.

```

```

125
126     Returns:
127         The dictionary representation of all messages matching the
128         above criteria that were created with this instance as the
129         parent. The newest 'count' messages (that are newer than time)
130         are retrieved and then returned in order such that the first
131         message in the returned list is the oldest.
132
133         Note that the first message returned is not necessarily the
134         oldest one that is newer than time. This can occur if the
135         number of matching messages is greater than 'count' since the
136         'count' newest are selected before their order is reversed.
137     """
138     return [message.to_dictionary() for message in
139             self.get_messages_query(message_type, recipient,
140                                     time = time).fetch(count)][::-1]]
141
142     def get_messages_query(self, message_type, recipient,
143                           time = datetime.min, sender = None,
144                           keys_only = False):
145         """ Return a message query from this instance.
146
147         Args:
148             message_type: The message type to retrieve. If left as None or
149                           the empty string then all messages matching other criteria
150                           will be returned.
151             recipient: The recipient of the messages to retrieve. All
152                       messages sent with a recipient of the empty string will also
153                       be retrieved.
154             time: All messages retrieved must have been created after this
155                  time.
156             sender: The sender of the message.
157             keys_only: If keys_only is set to true, this will only search
158                       for messages that have recipient = recipient. Thus, it will
159                       only include messages sent with no recipient if recipient
160                       is set to ''.
161
162         Returns:
163             A query object that can be fetched or further modified.
164         """
165         query = Message.all(keys_only = keys_only)
166         query.ancestor(self.key())
167         query.filter('date >', time)
168         if message_type is not None and message_type != '':
169             query.filter('msg_type =', message_type)
170         if sender:
171             query.filter('sender =', sender)
172         # Avoid doing two queries when we don't need to.
173         if recipient == '':
174             query.filter('recipient =', '')
175         else:
176             if keys_only:
177                 query.filter('recipient =', recipient)
178             else:

```

```

179         query.filter("recipient IN", [recipient, ''])
180     query.order('-date')
181     return query
182
183 def delete_messages(self, mtype = None):
184     """ Delete messages of a specified kind.
185
186     Args:
187         type: A string of the message type to delete.
188
189     Due to timeout issues with App Engine, this method will currently
190     only succeed when running on App Engine if the number of messages
191     being deleted is relatively small (~hundreds). It will attempt to
192     delete up to 1000. The timeout retry wrapper (see
193     game_server/autoretry_datastore.py) and using keys only search
194     drastically increases the chances of success, but this method is
195     still not guaranteed to complete.
196
197     For more information see:
198     http://groups.google.com/group/google-appengine/
199     http://stackoverflow.com/questions/108822/
200     delete-all-data-for-a-kind-in-google-app-engine
201     """
202     if mtype:
203         db.delete(Message.all(keys_only = True).filter('msg_type =',
204 mtype)
205                     .ancestor(self.key()).order('date').fetch(1000))
206         db.delete(Message.all(keys_only = True).ancestor(self.key()).order
207 ('date')
208                     .fetch(1000))
209 def check_player(self, pid):
210     """ Confirm that a player is currently in the instance.
211
212     Args:
213         pid: A string containing the player's email address.
214
215     Returns:
216         The email address of the player.
217
218     Raises:
219         ValueError if the player is not in this instance.
220     """
221     player = utils.check_playerid(pid)
222     if player in self.players:
223         return player
224     raise ValueError("%s is not in instance %s" % (pid, self.key().
225 name()))
226 def check_leader(self, pid):
227     """ Confirm that a player is the leader of the instance.
228
229     Args:

```

```

230     pid: A string containing the player's email address.
231
232     Returns:
233         The email address of the leader if pid contains it.
234
235     Raises:
236         ValueError if the player is not the leader of this instance.
237     """
238     player = utils.check_playerid(pid)
239     if player == self.leader:
240         return player
241     raise ValueError("You must be the leader to perform this operation
242 .")
243
244 def add_player(self, player):
245     """ Add a new player to this instance.
246
247     Args:
248         player: The email address of the player to add.
249
250     A player can join a game instance if it is not full and either the
251     instance is public or the player has been invited. If the player
252     is already in the game instance this will succeed without
253     modifying the instance.
254
255     Raises:
256         ValueError if the player is not already in the game and is
257         unable to join.
258     """
259     if player not in self.players:
260         if player not in self.invited and not self.public:
261             raise ValueError("%s not invited to instance %s."
262                             % (player, self.key().name()))
263         if self.full:
264             raise ValueError("%s could not join: instance %s is full"
265                             % (player, self.key().name()))
266         if player in self.invited:
267             self.invited.remove(player)
268         self.players.append(player)
269         self.set_full()

```

A.6: message.py - The message database model.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,

```



```

10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """ The Message database model class and associated methods."""
14
15 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
16
17 from datetime import datetime
18 from django.utils import simplejson
19 from game_server import iso8601
20 from google.appengine.ext import db
21
22 class Message(db.Expando):
23     """ A model for a message sent to a player in a game instance.
24
25     Messages are used to pass information from player to player and from
26     server to player. A Message's parent is the GameInstance which it is
27     created for.
28
29     Attributes:
30         msg_type: A string that acts as a key for the message.
31         recipient (optional): The intended recipient of this message.
32         content: JSON string that represents the contents of the message.
33         date: The date of creation, automatically set upon instantiation.
34         sender: A string describing the creator of the message.
35     """
36     msg_type = db.StringProperty(required=True)
37     recipient = db.StringProperty(required=False)
38     content = db.TextProperty(required=False)
39     date = db.DateTimeProperty(required=True, auto_now_add=True)
40     sender = db.StringProperty(required=True)
41
42     def to_dictionary(self):
43         """ Return a Python dictionary of the message.
44
45         Returns a dictionary of the message:
46             type: msg_type
47             mrec: recipient
48             contents: the Python representation of the content JSON string.
49             mtime: The iso8601 string representation of the creation time of
50                 the message.
51             msender: sender
52         """
53         return {'type' : self.msg_type, 'mrec' : self.recipient,
54                 'contents' : simplejson.loads(self.content),
55                 'mtime' : self.date.isoformat(),
56                 'msender' : self.sender}
57
58     def to_json(self):
59         """ Return a json representation of the dictionary of this message
60         . """
61         return simplejson.dumps(self.to_dictionary())

```

```

62 def get_content(self):
63     """ Return the Python representation of the contents of this
        message. """
64     return simplejson.loads(self.content)

```

A.1.2 Extensions

A.7: card_game.py - A library for handling card games in a game instance.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 Functionality for card games. Uses a single deck of cards for and
15 keeps track of each players hands using a dictionary stored with the
16 game instance.
17
18 The default deck is a standard 52 card deck. Each card is represented
19 as a two element list with its numerical value (1-13) as the first
20 element and its suit as the second element. Suits are the strings
21 'Hearts', 'Spades', 'Clubs' and 'Diamonds'.
22 """
23
24 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
25
26 import random
27 from django.utils import simplejson
28 from game_server.models.message import Message
29 from game_server.utils import get_boolean
30 from google.appengine.ext import db
31
32 default_deck = [[n, s] for n in range(14)[1:]
33                 for s in ['Hearts', 'Spades', 'Clubs', 'Diamonds']]
34
35 #####
36 # Server command functions #
37 #####
38
39 def set_deck_command(instance, player, arguments):
40     """ Set the instance's deck to a new list of cards.
41
42     Args:

```

```

43     instance: The GameInstance database model for this operation.
44     player: The email address of the player requesting the action.
45     For this command, the player must be the current leader of the
46     instance.
47     arguments: A list of the cards to set the deck to.
48
49 Resets the deck used by card games from a standard 52 card deck to
50 the deck specified by the arguments list. A new deck can only be set
51 when no other card game methods have been invoked for a particular
52 game instance. The deck will remain the same throughout the life of
53 the game instance.
54
55 Returns:
56     The number of cards in the new deck.
57
58 Raises:
59     A ValueError if the requesting player is not the leader of the
60     instance.
61 """
62 instance.check_leader(player)
63 return set_deck(instance, arguments)
64
65 def deal_cards_command(instance, player, arguments):
66     """ Deal cards to players.
67
68     Args:
69         instance: The GameInstance database model for this operation.
70         player: The email address of the player requesting the action.
71         The player must be the current leader of the instance.
72         arguments: A list of arguments to this command as explained below.
73
74     The arguments list for this command consists of five items in order:
75     1: cards_to_deal - The number of cards to deal as an integer.
76     2: shuffle_deck - A boolean controlling whether or not the deck
77         should be shuffled before the new hands are dealt. If the deck
78         is shuffled, all hands are also cleared regardless of the value
79         of is_new_hand.
80     3: is_new_hand - A boolean indicating whether this is a new hand
81         or not. If it is a new hand, then all hands will be cleared
82         before the new cards are dealt. If the deck is shuffled before
83         the cards are dealt then the hands are cleared automatically
84         and this has no effect.
85     4: ignore_empty_deck - Another boolean controlling whether to
86         ignore an empty deck or not. If it is true, then cards will be
87         dealt until the deck runs out and then this command will return
88         successfully. If it is false, an error will occur if the deck
89         runs out of cards.
90     5: A list of player id's to be dealt to in the order to deal to
91         them. Cards will be dealt one at a time to players in the order
92         that they appear in this list.
93
94     Cards are dealt to players using the instance's deck according to
95     the arguments specified above. The cards are dealt in the order
96     determined by the last shuffling. Until a deck is re-shuffled, cards

```

```

97     will be dealt as if they were removed from the top of the deck and
98     given to the player permanently.
99
100     Returns:
101         The hand of the requesting player after cards are dealt.
102
103     Raises:
104         An IndexError if the deck runs out of cards and empty deck errors
105         are not being ignored.
106         A ValueError if any of the player id's in the list of players to
107         deal to are not in the game instance.
108         A ValueError if the requesting player is not the leader of the
109         instance.
110     """
111     instance.check_leader(player)
112     cards_to_deal = int(arguments[0])
113     shuffle = get_boolean(arguments[1])
114     if shuffle:
115         shuffle_deck(instance)
116     is_new_hand = get_boolean(arguments[2])
117     ignore_empty_deck = get_boolean(arguments[3])
118     hands = deal_cards(instance, cards_to_deal, is_new_hand,
119                        ignore_empty_deck,
120                        arguments[4])
121     return hands[player]
122
123 def draw_cards_command(instance, player, arguments):
124     """ Draw cards from the deck and put them into the calling player's
125     hand.
126
127     Args:
128         instance: The GameInstance database model for this operation.
129         player: The email address of the player requesting the action.
130         arguments: A list of arguments to this command as explained below.
131
132     The arguments list for this command consists of two items in order:
133     1: cards_to_draw - The number of cards to attempt to draw.
134     2: ignore_empty_deck - A boolean controlling whether to ignore an
135     empty deck or not. If it is true, cards can be drawn until the
136     deck runs out and then this command will return
137     successfully. If it is false, an error will occur if the deck
138     runs out of cards and no changes will be made to the hand of
139     the player.
140
141     Returns:
142         The hand of the player after drawing the new cards.
143
144     Raises:
145         An IndexError if the deck runs out of cards and empty deck errors
146         are not being ignored.
147         ValueError if the requesting player is not in the instance.
148     """
149     cards_to_draw = int(arguments[0])
150     ignore_empty_deck = get_boolean(arguments[1])

```

```

149     return draw_cards(instance, player, cards_to_draw, ignore_empty_deck
150         )
151 def discard_command(instance, player, arguments):
152     """ Remove the specified cards from the calling player's hand.
153
154     Args:
155         instance: The GameInstance database model for this operation.
156         player: The email address of the player requesting the action.
157         arguments: A list of cards to discard.
158
159     Discarded cards are removed from a player's hand permanently. They
160     are not re-added to the deck of cards to be dealt to other
161     players. However, once a deck is shuffled, all cards become
162     available again including any that have been discarded.
163
164     If a player tries to discard a card that is not in their hand on the
165     server, the request to discard that particular card is ignored, but
166     the execution of the command continues.
167
168     Returns:
169         The current hand of the requesting player.
170     """
171     return discard(instance, player, arguments)
172
173 def pass_cards_to_player_command(instance, player, arguments):
174     """ Remove cards from the calling player's hand and add them to
175         another hand.
176
177     Args:
178         instance: The GameInstance database model for this operation.
179         player: The email address of the player requesting the action.
180         arguments: A list of two items. The first item is the email
181             address of the player to pass the cards to and the second is a
182             list of cards to pass to them.
183
184     Raises:
185         A ValueError if the player to pass the cards to is not in the game
186         instance.
187
188     """
189     hands = get_hand_dictionary(instance)
190     to_player = instance.check_player(arguments[0])
191     return pass_cards(instance, player, to_player, arguments[1])
192
193 def get_cards_remaining_command(instance, player, arguments = None):
194     """ Return the number of cards left in this deck to deal.
195
196     Args:
197         instance: The GameInstance database model for this operation.
198         player: The email address of the player requesting the action.
199         arguments: Not used, can be any value.
200
201     Returns:

```

```

201     The number of cards that can still be dealt before the deck is
202     empty. If the deck has not been set or no cards have been dealt
203     (in the case that the default deck is being used), returns -1.
204     """
205     return cards_left(instance)
206
207     #####
208     # Helpers #
209     #####
210
211     def get_deck(instance):
212         """ Return the deck for this instance.
213
214         Args:
215             instance: The GameInstance database model for this operation.
216
217         Returns:
218             The current deck for this instance. If no deck exists, returns the
219             default deck, unshuffled.
220         """
221         if 'crd_deck_index' not in instance.dynamic_properties():
222             instance.crd_deck_index = 0
223         if 'crd_deck' not in instance.dynamic_properties():
224             instance.crd_deck = [simplejson.dumps(card) for card in
225                                 default_deck]
226         return instance.crd_deck
227
228     def set_deck(instance, deck):
229         """ Set the deck for this instance to a new one.
230
231         Args:
232             instance: The GameInstance database model for this operation.
233             deck: A list of cards to set as a new deck.
234
235         Returns:
236             The number of cards in the new deck.
237
238         Raises:
239             AttributeError if a deck has already been created for this
240             instance.
241         """
242         if 'crd_deck' in instance.dynamic_properties():
243             raise AttributeError('Deck can only be set as the first operation
244                                 in '
245                                 'a card game.')
246         instance.crd_deck = [simplejson.dumps(card) for card in deck]
247         instance.crd_deck_index = 0
248         return len(instance.crd_deck)
249
250     def get_hand_dictionary(instance):
251         """ Return a dictionary with the hands of each player in the
252         instance.
253
254         Args:

```

```

252     instance: The GameInstance database model for this operation.
253
254 Returns:
255     A dictionary with a list for each player in the game. Each
256     player's list will include the cards currently in their hand. Keys
257     in the dictionary are the email addresses of players.
258     """
259     if 'crd_hands' not in instance.dynamic_properties():
260         return get_empty_hand_dictionary(instance)
261     else:
262         return simplejson.loads(instance.crd_hands)
263
264 def get_empty_hand_dictionary(instance):
265     """ Return a dictionary with an empty hand for each player in the
266         instance.
267
268     Args:
269         instance: The GameInstance database model for this operation.
270
271     Returns:
272         A dictionary with an empty list for each player in the game. Keys
273         in the dictionary are the email addresses of players.
274     """
275     hands = {}
276     for player in instance.players:
277         hands[player] = []
278     return hands
279
280 def set_hand_dictionary(instance, hands, send_messages = True):
281     """ Set the hands of all players and send new hand messages.
282
283     Args:
284         instance: The GameInstance database model for this operation.
285         hands: A dictionary containing the hand of each player in the
286         game.
287         send_messages: Whether or not to send a message to each player
288         with their new hand.
289
290     Stores the hands dictionary in the game instance. If send_messages
291     is True, this will also send a new 'crd_hand' message to each player
292     with their new hand.
293     """
294     if send_messages:
295         message_list = []
296         for player in instance.players:
297             message_list.append(instance.create_message(player, 'crd_hand',
298                                                         player, hands[player]
299                                                         )))
300         db.put(message_list)
301         instance.crd_hands = db.Text(simplejson.dumps(hands))
302
303 def get_player_hand(instance, player):
304     """ Get the hand of a single player in an instance.

```

```

304     Args:
305         instance: The GameInstance database model for this operation.
306         player: The email address of the player.
307
308     Returns:
309         The list of cards that the player has or an empty list if they
310         do not have a hand.
311
312     Raises:
313         ValueError if the player is not in the instance.
314     """
315     player = instance.check_player(player)
316     hands = get_hand_dictionary(instance)
317     return hands.get(player, [])
318
319 def set_player_hand(instance, player, hand, send_message = True):
320     """ Set the hand of a single player in an instance.
321
322     Args:
323         instance: The GameInstance database model for this operation.
324         player: The email address of the player.
325         hand: The new hand of the player.
326         send_message: Whether to send player a 'crd_hand' message
327             with their new hand.
328
329     Stores the new hands dictionary with the updated hand for player.
330     If send_message is True, a message will be sent to player with
331     their new hand.
332
333     Raises:
334         ValueError if the player is not in the instance.
335     """
336     player = instance.check_player(player)
337     hands = get_hand_dictionary(instance)
338     hands[player] = hand
339     set_hand_dictionary(instance, hands, send_messages = False)
340     if send_message:
341         instance.create_message(player, 'crd_hand', player, hand).put()
342
343 def get_next_card(instance):
344     """ Return the card in the deck.
345
346     Args:
347         instance: The GameInstance database model for this operation.
348
349     Returns:
350         The Python representation of the next card in the deck. Because
351         cards are stored as JSON strings they are first decoded before
352         being returned.
353
354     Raises:
355         ValueError if the JSON decoding fails.
356         IndexError if the deck has run out of cards.
357     """

```



```

358
359     if cards_left(instance) == 0:
360         raise IndexError('Deck is empty')
361     card = simplejson.loads(get_deck(instance)[instance.crd_deck_index])
362     instance.crd_deck_index = instance.crd_deck_index + 1
363     return card
364
365 def shuffle_deck(instance):
366     """ Shuffle the deck and reset all hands.
367
368     Args:
369         instance: The GameInstance database model for this operation.
370
371     Shuffles all cards in the original deck and makes them available to
372     be dealt or drawn again. Also clears all players hands.
373
374     Returns:
375         The number of cards in the deck.
376     """
377     deck = get_deck(instance)
378     random.shuffle(deck)
379     instance.crd_deck_index = 0
380     instance.crd_deck = deck
381
382     set_hand_dictionary(instance, get_empty_hand_dictionary(instance))
383     return len(instance.crd_deck)
384
385 def pass_cards(instance, from_player, to_player, cards):
386     """ Pass cards from one player to another.
387
388     Args:
389         instance: The GameInstance database model for this operation.
390         from_player: Email address of the player who is passing the cards.
391         to_player: Email address of the player who is receiving the cards.
392         cards: A list of cards to pass.
393
394     Searches the hand of from_player for each card in cards and if it
395     is present, transfers it to_player's hand. If a card is not present,
396     it is ignored.
397
398     Returns:
399         The hand of from_player after passing the cards.
400     """
401     hands = get_hand_dictionary(instance)
402     from_player = instance.check_player(from_player)
403     to_player = instance.check_player(to_player)
404
405     for card in cards:
406         try:
407             hands[from_player].remove(card)
408             hands[to_player].append(card)
409         except ValueError:
410             pass
411     set_hand_dictionary(instance, hands)

```

```

412     return hands[from_player]
413
414 def discard(instance, player, cards, send_message = True):
415     """ Remove the specified cards from player's hand.
416
417     Args:
418         instance: The GameInstance database model for this operation.
419         player: The email address of the player to discard cards from.
420         cards: The cards to be discarded.
421         send_message: Whether to send player a 'crd_hand' message
422             with their new hand.
423
424     Discarded cards are removed from a player's hand permanently. They
425     are not re-added to the deck of cards to be dealt to other
426     players. However, once a deck is shuffled, all cards become
427     available again including any that have been discarded.
428
429     If a player tries to discard a card that is not in their hand on the
430     server, the request to discard that particular card is ignored, but
431     the execution of the command continues.
432
433     Returns:
434         The hand of player after discarding the cards.
435
436     Raises:
437         ValueError if the player is not in the instance.
438     """
439     hand = get_player_hand(instance, player)
440     for card in cards:
441         try:
442             hand.remove(card)
443         except ValueError:
444             pass
445     set_player_hand(instance, player, hand, send_message)
446     return hand
447
448 def deal_cards(instance, cards_to_deal, is_new_hand, ignore_empty_deck
449     ,
450     deal_to):
451     """ Deal cards to players.
452
453     Args:
454         instance: The GameInstance database model for this operation.
455
456         cards_to_deal: The number of cards to deal as an integer.
457         is_new_hand: A boolean indicating whether this is a new hand or
458             not. If it is a new hand, then all hands will be cleared before
459             the new cards are dealt. If the deck is shuffled before the
460             cards are dealt then the hands are cleared automatically and
461             this has no effect.
462         ignore_empty_deck: Another boolean controlling whether to ignore
463             an empty deck or not. If it is true, then cards will be dealt
464             until the deck runs out and then this command will return
465             successfully. If it is false, an error will occur if the deck

```

```

465         runs out of cards.
466     deal_to: A list of player id's to be dealt to in the order to deal
467         to them. Cards will be dealt one at a time to players in the
468         order that they appear in this list.
469
470     The cards are dealt in the order determined by the last
471     shuffling. Until a deck is re-shuffled, cards will be dealt as if
472     they were removed from the top of the deck and given to the player
473     permanently.
474
475     Returns:
476         The hand of the requesting player after cards are dealt.
477
478     Raises:
479         ValueError if a player in deal_to is not in the instance.
480         IndexError if the deck runs out of cards and ignore_empty_deck is
481         not True.
482     """
483     hands = {}
484     if not is_new_hand:
485         hands = get_hand_dictionary(instance)
486
487     if cards_to_deal:
488         deal_to = [instance.check_player(pid) for pid in deal_to]
489         for player in deal_to:
490             hands.setdefault(player, [])
491         try:
492             for i in xrange(cards_to_deal):
493                 for player in deal_to:
494                     hands[player].append(get_next_card(instance))
495         except IndexError:
496             if not ignore_empty_deck:
497                 raise
498
499     set_hand_dictionary(instance, hands)
500     return hands
501
502 def draw_cards(instance, player, cards_to_draw,
503               ignore_empty_deck = True, send_message = True):
504     """ Draw cards from the deck and put them into player's hand.
505
506     Args:
507         instance: The GameInstance database model for this operation.
508         player: The email address of the player to give the cards to.
509         cards_to_draw: The number of cards to draw from the deck.
510         ignore_empty_deck - A boolean controlling whether to ignore an
511             empty deck or not. If it is true, cards can be drawn until the
512             deck runs out and then this command will return
513             successfully. If it is false, an error will occur if the deck
514             runs out of cards and no changes will be made to the hand of
515             the player.
516         send_message: Whether to send player a 'crd_hand' message
517             with their new hand.
518

```

```

519     Returns:
520         The hand of the requesting player after they have drawn their
521         cards.
522
523     Raises:
524         ValueError if player is not in the game instance.
525         IndexError if the deck runs out of cards and ignore_empty_deck is
526         not True.
527     """
528     hand = get_player_hand(instance, player)
529     try:
530         for i in xrange(cards_to_draw):
531             hand.append(get_next_card(instance))
532     except IndexError:
533         if not ignore_empty_deck:
534             raise
535     set_player_hand(instance, player, hand, send_message)
536     return hand
537
538 def cards_left(instance):
539     """ Return the number of cards left to deal before a shuffle is
540         required.
541
542     Args:
543         instance: The GameInstance database model for this operation.
544
545     Returns
546         The number of cards that can still be dealt before the deck is
547         empty. If the deck has not been set or no cards have been dealt
548         (in the case that the default deck is being used), returns -1.
549     """
549     if 'crd_deck' not in instance.dynamic_properties():
550         return -1
551     return len(instance.crd_deck) - instance.crd_deck_index

```

A.8: scoreboard.py - A library for keeping track of a per instance scoreboard.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 Provides methods to use a scoreboard with GameInstance models.
15

```

```

16 All scores are stored as integers with the convention that higher
17 numbers are better.
18 """
19
20 __authors__ = ["Bill Magnuson" <billmag@mit.edu>']
21
22 import operator
23 from django.utils import simplejson
24 from google.appengine.ext import db
25
26 #####
27 # Server command functions #
28 #####
29
30 def get_scoreboard_command(instance, player, arguments = None):
31     """ Get the current scoreboard.
32
33     Args:
34         instance: The GameInstance database model for this operation.
35         player: The email address of the player requesting this action.
36         arguments: Not used, can be any value.
37
38     Returns:
39         The complete scoreboard as a list of [score, email] lists for
40         each player in the game. The scoreboard is sorted with the highest
41         score first.
42     """
43     return format_scoreboard_for_app_inventor(get_scoreboard(instance))
44
45 def get_score_command(instance, player, arguments):
46     """ Set the score of a single player.
47
48     Args:
49         instance: The GameInstance database model for this operation.
50         player: The email address of the player requesting this action.
51         arguments: A one item list containing the player id of the
52         player to get the score of.
53
54     Returns:
55         The score of the requested player.
56
57     Raises:
58         ValueError if the player in arguments is not in the game.
59     """
60     get_score_player = instance.check_player(arguments[0])
61     return get_score(instance, get_score_player)
62
63 def set_score_command(instance, player, arguments):
64     """ Set a player's score to a new value.
65
66     Args:
67         instance: The GameInstance database model for this operation.
68         player: The email address of the player to set the score of.
69         arguments: A list of two items. The first item is the player id

```

```

70         of the player who's score is to be set. The second item is the
71         integer to set that player's score to.
72
73     Returns:
74         The complete scoreboard after setting the new score value.
75
76     Raises:
77         ValueError if the specified player is not in the instance.
78     """
79     player = instance.check_player(arguments[0])
80     new_score = arguments[1]
81     board = set_score(instance, player, new_score)
82     return format_scoreboard_for_app_inventor(board)
83
84
85 def add_to_score_command(instance, player, arguments):
86     """ Change a player's score by an integer amount.
87
88     Args:
89         instance: The GameInstance database model for this operation.
90         player: The email address of the player to add points to.
91         arguments: A list of two items. The first item is the player id
92             of the player who's score is to be set. The second item is the
93             integer amount to change that player's score by. This value can
94             be positive or negative.
95
96     In order for this operation to work correctly scores must be
97     represented in the scoreboard as single integer items.
98
99     Returns:
100         The complete scoreboard after adding to player's score.
101
102     Raises:
103         ValueError if the specified player is not in the instance.
104         ValueError if the specified score cannot parse correctly.
105     """
106     player = instance.check_player(arguments[0])
107     delta = int(arguments[1])
108     board = add_to_score(instance, player, delta)
109     return format_scoreboard_for_app_inventor(board)
110
111 def clear_scoreboard_command(instance, player, arguments = None):
112     """ Reset all scores to 0.
113
114     Args:
115         instance: The GameInstance database model for this operation.
116         player: The email address of the player requesting this action.
117             For this command, the player must be the current leader of the
118             instance.
119         arguments: Not used, can be any value.
120
121     Returns:
122         An empty scoreboard with a score of 0 for each player.
123

```

```

124     Raises:
125         ValueError if player is not the leader of this instance.
126     """
127     instance.check_leader(player)
128     instance.scoreboard = '{}'
129     return get_scoreboard_command(instance, player, arguments)
130
131
132 #####
133 # Helpers #
134 #####
135
136 def get_score(instance, player):
137     """ Get a player's score.
138
139     Args:
140         instance: The instance to get the scoreboard from.
141         player: The player to check the score of.
142
143     Returns:
144         The players score.
145
146     Raises:
147         ValueError if the player is not in the instance.
148     """
149     player = instance.check_player(player)
150     board = get_scoreboard(instance)
151     return board[player]
152
153 def set_score(instance, player, new_score):
154     """ Set a player's score.
155
156     Args:
157         instance: The game instance to modify the scoreboard of.
158         player: The player to set the score of.
159         new_score: An integer to set their score to.
160
161     Returns:
162         The scoreboard as a dictionary after setting a new value for
163         player's score.
164
165     Raises:
166         ValueError if the player is not in the instance.
167     """
168     player = instance.check_player(player)
169     scoreboard = get_scoreboard(instance)
170     scoreboard[player] = new_score
171     instance.scoreboard = simplejson.dumps(scoreboard)
172     return scoreboard
173
174 def add_to_score(instance, player, delta):
175     """ Change a player's score by delta.
176
177     Args:

```

```

178     instance: The game instance to modify the scoreboard of.
179     player: The player to change the score of.
180     delta: The integer amount to change player's score by (can be
181     negative).
182
183     In order for this operation to work correctly scores must be
184     represented in the scoreboard as single integer items.
185
186     Returns:
187     The scoreboard as a dictionary after modifying player's score.
188     """
189     player = instance.check_player(player)
190     scoreboard = get_scoreboard(instance)
191     if player in scoreboard:
192         scoreboard[player] += delta
193     else:
194         scoreboard[player] = delta
195     instance.scoreboard = db.Text(simplejson.dumps(scoreboard))
196     return scoreboard
197
198 def get_scoreboard(instance):
199     """ Get a dictionary of the scoreboard for the specified instance.
200
201     Args:
202     instance: The instance to get the scoreboard from.
203
204     Returns:
205     A dictionary with a score entry for each player in the
206     instance. If no score was previously present, a value of
207     0 is entered.
208     """
209     board = None
210     if 'scoreboard' not in instance.dynamic_properties():
211         board = {}
212     else:
213         board = simplejson.loads(instance.scoreboard)
214     for player in instance.players:
215         if not board.has_key(player):
216             board[player] = 0
217     return board
218
219 def format_scoreboard_for_app_inventor(board):
220     """ Return a scoreboard suitable to return to App Inventor.
221
222     Args:
223     board: The dictionary of scores for all players in the game.
224
225     Returns:
226     A list of [score, player email] lists ordered by highest score.
227     """
228     board_list = [[v,k] for k, v in board.items()]
229     board_list.sort(key = operator.itemgetter(0), reverse = True)
230     return board_list

```

A.2 Custom Modules

A.9: commands.py - The command dictionary for custom modules.

```
1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #      http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 Defines the commands available from custom modules. Custom modules
15 differ from built in server commands or server extensions because they
16 are more narrowly focused on a particular game's functionality.
17
18 Custom modules will generally be built on a per game basis and
19 included when game creators deploy their own App Engine servers.
20
21 This file currently includes commands for custom modules meant to be
22 used as examples. These can be removed to decrease load time if they
23 are not being used.
24 """
25
26 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
27
28 from custom_modules.androids_to_androids import ata_commands
29 from custom_modules.bulls_and_cows import bac_commands
30 from custom_modules.amazon import amazon_commands
31 from custom_modules.voting import voting_commands
32
33 custom_command_dict = {
34     # Androids to Androids
35     'ata_new_game' : ata_commands.new_game_command,
36     'ata_submit_card' : ata_commands.submit_card_command,
37     'ata_end_turn' : ata_commands.end_turn_command,
38
39     # Bulls and Cows
40     'bac_new_game' : bac_commands.new_game_command,
41     'bac_guess' : bac_commands.guess_command,
42
43     # Amazon
44     'amz_keyword_search' : amazon_commands.keyword_search_command,
45     'amz_isbn_search' : amazon_commands.isbn_search_command,
46
47     # Voting
48     'vot_cast_vote' : voting_commands.cast_vote_command,
```

```

49     'vot_get_results' : voting_commands.get_results_command,
50     'vot_new_poll' : voting_commands.make_new_poll_command,
51     'vot_close_poll' : voting_commands.close_poll_command,
52     'vot_delete_poll' : voting_commands.delete_poll_command,
53     'vot_get_poll_info' : voting_commands.get_poll_info_command,
54     'vot_get_my_polls' : voting_commands.get_my_polls_command
55 }

```

A.2.1 Amazon

A.10: amazon.commands.py - Amazon server commands.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """ Looks up books on Amazon by keyword or ISBN
14
15 Uses a library originally downloaded from
16 http://blog.umlungu.co.uk/blog/2009/jul/12/pyaws-adding-request-
    authentication/
17 to access AWS E-Commerce Service API's and retrieve book results
18 for searches by keyword and ISBN number.
19 """
20
21 __authors__ = ["Bill Magnuson" <billmag@mit.edu>',
22               "Dave Wolber" <wolber@usfca.edu>']
23
24
25 # This file has had its license and secret keys removed and will not
    function.
26 license_key = ''
27 secret_key = ''
28
29 from pyaws import ecs
30
31 return_limit = 5
32
33 def keyword_search_command(model, player, arguments):
34     """ Return books by keyword.
35
36     Args:
37         model: Not used, can be anything.

```

```

38     player: Not used, can be anything.
39     arguments: A one item list containing the keywords to search for.
40
41     Returns:
42     A list of three item lists. Each sublist represents
43     a result and includes the book title, its lowest found
44     price and its ASIN number.
45     """
46     return amazon_by_keyword(arguments[0])
47
48 def isbn_search_command(model, player, arguments):
49     """ Return a book result by ISBN number.
50
51     Args:
52     model: Not used, can be anything.
53     player: Not used, can be anything.
54     arguments: A one item list containing the keywords to search for.
55
56     Returns:
57     A list with a single sublist representing the book found.
58     The sublist contains the book title, its lowest found
59     price and its ASIN number.
60
61     Raises:
62     ValueError if the ISBN number is invalid.
63     """
64     return amazon_by_isbn(arguments[0])
65
66 def amazon_by_keyword(keyword):
67     """ Use the ecs library to search for books by keyword.
68
69     Args:
70     keyword: A string of keyword(s) to search for.
71
72     Returns:
73     A list of three item lists. Each sublist represents
74     a result and includes the book title, its lowest found
75     price and its ASIN number.
76     """
77     ecs.setLicenseKey(license_key)
78     ecs.setSecretKey(secret_key)
79     ecs.setLocale('us')
80
81     books = ecs.ItemSearch(keyword, SearchIndex='Books', ResponseGroup='
      Medium')
82     return format_output(books)
83
84 def amazon_by_isbn(isbn):
85     """ Use the ecs library to search for books by ISBN number.
86
87     Args:
88     isbn: The 10 digit ISBN number to look up.
89
90     Returns:

```

```

91     A list with a single sublist representing the book found.
92     The sublist contains the book title, its lowest found
93     price and its ASIN number.
94
95     Raises:
96         ValueError if the ISBN number is invalid.
97     """
98     ecs.setLicenseKey(license_key)
99     ecs.setSecretKey(secret_key)
100    ecs.setLocale('us')
101    try:
102        books = ecs.ItemLookup(isbn, IdType='ISBN', SearchIndex='Books',
103                               ResponseGroup='Medium')
104        return format_output(books)
105    except ecs.InvalidParameterValue:
106        raise ValueError('Invalid ISBN')
107
108    def format_output(books):
109        """ Return a formatted output list from an iterator returned by ecs.
110
111        Args:
112            books: An iterator of book results from the ecs library.
113
114        Returns:
115            A list of three item lists. Each sublist represents
116            a result and includes the book title, its lowest found
117            price and its ASIN number.
118        """
119        size = min(len(books), return_limit)
120        return [[books[i].Title, get_amount(books[i]), books[i].ASIN]
121                for i in xrange(size)]
122
123    def get_amount(book):
124        """ Return the lowest price found or 'Not found.' if none exists.
125        """
126        try:
127            if book.OfferSummary and book.OfferSummary.LowestNewPrice:
128                return book.OfferSummary.LowestNewPrice.FormattedPrice
129        except:
130            return 'Not found.'

```

A.2.2 Androids to Androids

A.11: ata_commands.py - Androids to Androids game commands.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7

```

```

8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 A set of server commands to implement Androids to Androids.
15
16 Androids to Androids is a card game played by at least three
17 players. The first leader of the game is the creator of the game
18 instance. The game proceeds in rounds with the winner of each round
19 becoming the leader of the next round. To start the game, each player
20 is dealt seven cards with nouns on them. These cards comprise their
21 hand.
22
23 At the beginning of each round, an adjective or characteristic is
24 chosen at random and sent to each player. Every player except the
25 leader will then choose a card from their hand to submit for the
26 round. Upon submission their hand will be replenished with another
27 card so that they always have seven cards in their hand.
28
29 The leader will then choose a single noun card from those submitted by
30 the other players in response to the characteristic. The leader can
31 use any criteria they wish to select the card that should win the
32 round, however, they are not allowed to know the identity of the
33 person that submits each card.
34
35 Once a winner is chosen, a new round is started with the previous
36 winner as the new leader. Play continues in this way until one of the
37 players reaches a predetermined winning score and is declared the
38 winner.
39
40 Each command returns information that is immediately useful to the
41 player who requested the command. In addition, any changes to their
42 hand or the set of cards players have submitted will be sent to them
43 via message so that they can easily recover state if they lose their
44 active session in the game.
45
46 Submitting cards and ending turns both require that the player submit
47 the round number that they intend for that action to apply to. If
48 that number does not match the current round the action will be
49 ignored and the command will return information to allow that player
50 to get back up to date with the game.
51 """
52
53 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
54
55 import random
56 import decks
57 from game_server.extensions import scoreboard
58 from game_server.extensions import card_game
59 from google.appengine.ext import db
60 from django.utils import simplejson

```

```

61
62 hand_size = 7
63 winning_score = 5
64 min_players = 3
65
66 #####
67 # Game Commands #
68 #####
69
70 def new_game_command(instance, player, arguments = None):
71     """ Start a new game of Androids to Androids.
72
73     Args:
74         instance: The GameInstance database model for this operation.
75         player: The player starting the game. Must be the current leader
76                 of the instance instance.
77         arguments: Not used, can be any value.
78
79     Closes the game to new players, deals a new hand to each player and
80     selects a new characteristic card to begin round 1. Sends a new
81     game message to all players with the starting card and the empty
82     scoreboard.
83
84     Each player will also receive a message of type crd_hand that
85     contains all of the cards dealt to them.
86
87     Returns:
88         A three item list consisting of the new characteristic card for
89         this turn, the current (empty) scoreboard, and the requesting
90         player's current hand.
91
92     Raises:
93         ValueError if an Androids to Androids game is already in
94         progress, if player is not the current leader of the game or if
95         there are not enough players in the game to begin.
96     """
97     player = instance.check_leader(player)
98     instance.public = False
99
100     if 'ata_round' in instance.dynamic_properties():
101         raise ValueError("This game is already in progress. " +
102                           "Please refresh the game state.")
103
104     if len(instance.players) < min_players:
105         raise ValueError("Androids to Androids requires at least %d
106                           % min_players)
107
108     instance.max_players = len(instance.players)
109     try:
110         card_game.set_deck(instance, decks.noun_cards)
111     except AttributeError:
112         pass
113     card_game.shuffle_deck(instance)

```

```

114     hands = card_game.deal_cards(instance, hand_size, True, False,
115                                   instance.players)
116
117     instance.starting_players = instance.players
118     instance.ata_round = 0
119     setup_new_round(instance)
120     board = scoreboard.clear_scoreboard_command(instance, player)
121     instance.create_message(instance.leader, 'ata_new_game', '',
122                               [instance.ata_char_card, board]).put()
123
124     return [instance.ata_char_card, board, hands[player]]
125
126 def submit_card_command(instance, player, arguments):
127     """ Submit a noun card for the current round.
128
129     Args:
130         instance: The GameInstance database model for this operation.
131         player: The player submitting the card. Cannot be the leader.
132         arguments: A two item list consisting of the round to submit this
133             card for and the card itself.
134
135     If the submission is for the wrong round, a four item list with an
136     error string as its first element will be returned. The remaining
137     elements are the player's hand, the current round and the current
138     characteristic card to respond to. No other action will be taken.
139
140     Removes the indicated card from the player's hand and adds it
141     to this round's submissions. The current submissions are sent via
142     message to all players.
143
144     The requesting player's hand will be dealt another card after
145     removing the submitted one. The updated hand will be sent to the
146     requesting player in a message and be included in the return value
147     of this command.
148
149     Returns:
150         If the submission is for the correct round, returns a three item
151         list consisting of the current round number, a list of the
152         submissions made so far by other players in this round and the
153         player's new hand.
154
155     Raises:
156         ValueError if player is the leader. The leader is not allowed to
157         submit cards.
158     """
159     if int(arguments[0]) != instance.ata_round:
160         hand = card_game.get_player_hand(instance, player)
161         return ['You tried to submit a card for the wrong round. ' +
162                'Please try again.', hand, instance.ata_round,
163                instance.ata_char_card]
164
165     missing_player = check_players(instance)
166     if missing_player:
167         return missing_player

```

```

168
169     if player == instance.leader:
170         raise ValueError("The leader may not submit a card.")
171
172     submission = arguments[1]
173     submissions = set_submission(instance, player, submission).values()
174     instance.create_message(player, 'ata_submissions', '',
175                             [instance.ata_round, submissions, submission
176                               ]).put()
177
178     card_game.discard(instance, player, [submission], False)
179     hand = card_game.draw_cards(instance, player, 1)
180     return [instance.ata_round, submissions, hand]
181
182 def end_turn_command(instance, player, arguments):
183     """ End the current turn and start a new one.
184
185     Args:
186         instance: The GameInstance database model for this operation.
187         player: The player submitting the card. Must be the current
188         leader.
189         arguments: A two item list consisting of the round number to end
190         and the selected winning card.
191
192     If the command is for the wrong round, a four item list with an
193     error string as its first element will be returned. The remaining
194     elements are the player's hand, the current round and the current
195     characteristic card to respond to. No other action will be taken.
196
197     Ends the current turn and adds 1 point to the score of the player
198     who submitted the winning card. If that player has reached the
199     winning score, an 'ata_game_over' message will be sent to all
200     players. The game over message content will be a three item list as
201     its contents. The list contains the final round number, the winning
202     card and the final scoreboard.
203
204     Otherwise, sends an 'ata_new_round' message to all players. The new
205     round message contents will be a five item list with the round
206     number, the new characteristic card, the previous round winner, the
207     winning card and the current scoreboard.
208
209     Returns:
210         If the command was for the correct round, returns the content of
211         whichever message was sent to all players as described above.
212
213     Raises:
214         ValueError if player is not the leader.
215         KeyError if no player has submitted the winning card.
216     """
217     if int(arguments[0]) != instance.ata_round:
218         hand = card_game.get_player_hand(instance, player)
219         return ['You tried to end a turn that has already ended. ' +
220               'Please try again.', hand, instance.ata_round,
221               instance.ata_char_card]

```



```

221 missing_player = check_players(instance)
222 if missing_player:
223     return missing_player
224
225 instance.check_leader(player)
226 card = arguments[1]
227 winner = None
228 for player, submitted_card in get_submissions_dict(instance).items():
229     if card == submitted_card:
230         winner = player
231         break
232 if winner == None:
233     raise KeyError('No player has submitted the card %s.' % card)
234 board = scoreboard.add_to_score(instance, winner, 1)
235
236 # Check to see if anyone has won
237 instance.leader = winner
238 if board[winner] == winning_score:
239     return end_game(instance, card)
240
241 setup_new_round(instance)
242 return_scoreboard = scoreboard.format_scoreboard_for_app_inventor(
243     board)
244 content = [instance.ata_char_card, return_scoreboard,
245            instance.ata_round, winner, card]
246 instance.create_message(instance.leader, 'ata_new_round', '',
247                          content).put()
248 return content
249
250 #####
251 # Helpers #
252 #####
253
254 def check_players(instance):
255     """ Checks to see if any of the starting players have left.
256
257     Args:
258         instance: The GameInstance model for this operation.
259
260     If a player has left the game, they are invited back and
261     a ValueError is raised.
262
263     Raises:
264         ValueError if a player has left the game.
265     """
266     if len(instance.players) < len(instance.starting_players):
267         for starting_player in instance.starting_players:
268             if starting_player not in instance.players:
269                 instance.invited.append(starting_player)
270                 return ('%s left during your game. They have ' %
271                        starting_player +
272                        'been invited and must rejoin before continuing.')
273     return False

```

```

272
273 def end_game(instance, winning_card):
274     """ End the current game and inform all players of the winner.
275
276     Args:
277         instance: The GameInstance database model for this operation.
278         winning_card: The card chosen as the winner of the final round.
279
280     Sends an 'ata_game_over' message from the winner to all players with
281     a three item list as its contents. The list contains the final round
282     number, the winning card and the final scoreboard.
283
284     Deletes the ata_round, ata_char_card and ata_submissions properties
285     from the GameInstance database model to allow for a new game to be
286     player in this same instance with the previous winner as the new
287     leader.
288
289     Returns:
290         The content of the message sent to all players.
291     """
292     content = [instance.ata_round, winning_card,
293               scoreboard.get_scoreboard_command(instance, instance.
294               leader)]
295     instance.create_message(instance.leader, 'ata_game_over', '',
296                             content).put()
297     del instance.ata_round
298     del instance.ata_char_card
299     del instance.ata_submissions
300     instance.scoreboard = '{}'
301     return content
302
303 def setup_new_round(instance):
304     """ Update the round number, char card and submissions for a new
305     round.
306
307     Args:
308         instance: The GameInstance database model for this operation.
309
310     Increments ata_round, clears the submissions dictionary and sets
311     ata_char_card to a new value from the list of characteristic cards.
312     """
313     instance.ata_round += 1
314     instance.ata_submissions = db.Text('{}')
315
316     new_card = random.choice(decks.characteristic_cards)
317     if 'ata_char_card' in instance.dynamic_properties():
318         while instance.ata_char_card == new_card:
319             new_card = random.choice(decks.characteristic_cards)
320     instance.ata_char_card = new_card
321
322 def get_submissions_dict(instance):
323     """ Return a Python dictionary that maps cards to players.
324
325     Args:

```

```

323     instance: The GameInstance database model for this operation.
324
325     Returns:
326         A Python dictionary of cards to players for all cards
327         submitted so far during this round.
328     """
329     return simplejson.loads(instance.ata_submissions)
330
331 def set_submission(instance, player, card):
332     """ Records the submission of card as coming from player.
333
334     Args:
335         instance: The GameInstance database model for this operation.
336         player: The player submitting the card.
337         card: The card to submit.
338
339     Returns:
340         A Python dictionary of cards to players for all cards
341         submitted so far during this round.
342     """
343     submissions = get_submissions_dict(instance)
344     if player in submissions:
345         raise ValueError('You have already submitted a card for this round
346         .')
347     submissions[player] = card
348     instance.ata_submissions = simplejson.dumps(submissions)
349     return submissions

```

A.2.3 Bulls and Cows

A.12: bac_commands.py - Bulls and Cows game commands.

```

1  # Copyright 2010 Google Inc.
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #     http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """
14 A version of bulls and cows using colors.
15
16 At the beginning of a new game a solution sequence of four colors is
17 randomly chosen from the set of colors. Each color appears at most
18 once in the solution. The player then makes guesses on the sequence
19 of colors in the solution. After each guess, they are informed of how

```

```

20 many 'cows' and 'bulls' they have in their guess. A 'bull' is when a
21 player has the correct color in the correct position in their guess. A
22 'cow' is when a player has one of the correct colors, but it is in
23 the wrong position.
24
25 Although the solution only includes each color once, a player is
26 allowed to use the same color more than once in their guess. While
27 obviously not correct, doing so might give the player information
28 that they want about the solution.
29
30 The player begins with a score such that they will end with a score
31 of zero if they guess completely wrong every time. After each guess
32 is made, two points are deducted for each completely wrong color and
33 one point is deducted for a correct color in the wrong spot (a
34 cow). No points are deducted for a bull. If a player does not
35 determine the correct sequence before they run out of guesses they
36 are not awarded a score. """
37
38 __authors__ = ["Bill Magnuson" <billmag@mit.edu>"]
39
40 from random import sample
41 from django.utils import simplejson
42 from game_server.extensions import scoreboard
43 from game_server.models.message import Message
44 from google.appengine.ext import db
45 from google.appengine.ext.db import Key
46
47 starting_guesses = 12
48 solution_size = 4
49 colors = ['Blue', 'Green', 'Orange', 'Red', 'Yellow', 'Pink']
50
51 def new_game_command(instance, player, arguments = None):
52     """ Start a new game and reset any game in progress.
53
54     Args:
55         instance: The GameInstance database model for this operation.
56         player: The player starting a new game. Must be the only player
57             in the instance.
58         arguments: Not used, can be any value.
59
60     Returns:
61         A list containing the number of guesses remaining, the starting
62         score of the player, the player's historical high score and the
63         number of games completed in the past.
64
65     Raises:
66         ValueError if there is more than 1 player in the instance
67         or the player is not the current leader.
68     """
69     old_games = instance.get_messages_query('bac_game', player,
70                                             sender = player,
71                                             keys_only = True)
72     db.delete(old_games)
73

```

```

74     score = scoreboard.get_score(instance, player)
75     if (score == 0):
76         # Score is [high score, total score, games played]
77         score = [0, 0, 0]
78         scoreboard.set_score(instance, player, score)
79
80     game = Message(parent = instance, sender = player,
81                   msg_type = 'bac_game', recipient = player)
82     game.bac_solution = sample(colors, solution_size)
83     game.bac_guesses_remaining = starting_guesses
84     game.bac_score = solution_size * starting_guesses * 2
85     game.bac_last_guess = ['']
86     game.bac_last_reply = ''
87     game.put()
88
89     return [game.bac_guesses_remaining, game.bac_score, score,
90           game.key().id()]
91
92 def guess_command(instance, player, arguments):
93     """ Evaluate a guess and determine the score.
94
95     Args:
96         instance: The GameInstance database model for this operation.
97         player: The player making the guess. Must be the leader of
98             the instance.
99         arguments: A two element list containing the game id and a second
100             list with the guessed colors.
101
102     new_game_command must be invoked before a guess can be made.
103
104     Returns:
105         If the player has guessed correctly:
106             A two element list containing a score list and a boolean of
107             whether or not this game set a new high score. The score list is
108             a three element list containing the player's high score, their
109             total score and their total number of games played.
110
111         Otherwise:
112             A four element list containing the player's remaining score, the
113             number of guesses remaining, the number of bulls for this guess
114             and the number of cows for this guess.
115
116     Raises:
117         ValueError if the player is not the current instance leader and
118         only member of the game.
119         ValueError if the player has no guesses remaining.
120         ValueError if the guess does not have the correct number of
121         elements.
122         ValueError if no game has been started yet.
123     """
124     guess = arguments[1]
125     if len(guess) != solution_size:
126         raise ValueError("Guess was not the right number of elements.")
127

```

```

128 game = db.get(Key.from_path('Message', int(arguments[0]),
129                             parent = instance.key()))
130
131 if game is None:
132     raise ValueError("Game not found. Please start a new game.")
133 if game.sender != player:
134     raise ValueError("This is not your game. Please start a new game."
135 )
136
137 if guess == game.bac_last_guess:
138     return simplejson.loads(game.bac_last_reply)
139
140 if game.bac_guesses_remaining == 0:
141     raise ValueError("No turns left, please start a new game.")
142
143 return_content = None
144
145 if guess == game.bac_solution:
146     game.bac_guesses_remaining = 0
147     new_high_score = False
148     score = scoreboard.get_score(instance, player)
149     if game.bac_score > score[0]:
150         new_high_score = True
151         score[0] = game.bac_score
152         score[1] = score[1] + game.bac_score
153         score[2] = score[2] + 1
154         scoreboard.set_score(instance, player, score)
155         return_content = [score, new_high_score]
156 else:
157     game.bac_guesses_remaining -= 1
158     bulls = cows = 0
159     for i in xrange(solution_size):
160         if guess[i] == game.bac_solution[i]:
161             bulls += 1
162         elif guess[i] in game.bac_solution:
163             cows += 1
164
165     score_deduction = solution_size * 2 - cows - 2 * bulls
166     game.bac_score -= score_deduction
167     return_content = [game.bac_guesses_remaining, game.bac_score,
168                     bulls, cows]
169
170 game.bac_last_reply = simplejson.dumps(return_content)
171 game.bac_last_guess = guess
172 game.put()
173 return return_content

```

A.2.4 Voting

A.13: voting.commands.py - Voting commands.

```

1 # Copyright 2010 Google Inc.
2 # Licensed under the Apache License, Version 2.0 (the "License");

```

```

3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5
6  #      http://www.apache.org/licenses/LICENSE-2.0
7
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
11 # See the License for the specific language governing permissions and
12 # limitations under the License.
13 """ Commands for a voting application.
14
15 The commands are split into two categories. The first is for people
16 who are performing the voting. The second category is for the
17 creation and management of polls.
18
19 Voting:
20 Players find out about new polls by retrieving messages with types
21 'poll' or 'closed_poll' from the instance.
22
23 Once a player has found out about polls, they can cast votes and
24 get results for closed polls and polls they have already voted in.
25
26 When a player votes in a poll they immediately receive the current
27 results of that poll. They will be able to fetch those results until
28 the poll creator deletes the poll.
29
30 Poll Management:
31 The remaining commands are for managing polls. Polls can be
32 created, closed and deleted. Players can get the polls they have
33 created with the get my polls command.
34 """
35
36 __authors__ = ["Bill Magnuson" <billmag@mit.edu>']
37
38 from django.utils import simplejson
39 from game_server.models.message import Message
40 from google.appengine.ext import db
41
42 def cast_vote_command(instance, player, arguments):
43     """ Cast a vote in a poll and return its current results.
44
45     Args:
46         instance: The parent GameInstance model of this poll.
47         player: The player that is casting a vote.
48         arguments: A two item list of the poll id and the zero
49             based index of the option to select.
50
51     Returns:
52         A two item list containing a message and the current votes
53         for the poll. The message will be one of:
54         Your vote was already counted in this poll.
55         Poll closed to new votes.

```

```

56         Vote accepted.
57
58     Raises:
59         ValueError if the vote index is larger than the number
60         of options.
61         ValueError if the player is not in the instance.
62     """
63     instance.check_player(player)
64     poll = get_poll(instance, arguments[0])
65     if not poll.open:
66         return ['Poll closed to new votes.', poll.votes]
67     if player in poll.voters:
68         return ['Your vote was already counted in this poll.', poll.votes]
69
70     try:
71         poll.voters.append(player)
72         vote_index = int(arguments[1])
73         poll.votes[vote_index] += 1
74         poll.put()
75     except ValueError:
76         raise ValueError('Invalid vote choice.')
77     return ['Vote accepted.', poll.votes]
78
79 def get_results_command(instance, player, arguments):
80     """ Gets the results of a poll.
81
82     Args:
83         instance: The parent GameInstance model of the poll.
84         player: The player requesting the results.
85         arguments: A one item list containing the id number of the poll.
86
87     Returns:
88         If the player has not voted in this poll and it is still open,
89         this will return a single item list with a message for the
90         requesting player.
91         Otherwise returns a list with information about the poll. See
92         get_poll_return_list for its format.
93
94     Raises:
95         ValueError if the player is not in the instance.
96     """
97     instance.check_player(player)
98     poll = get_poll(instance, arguments[0])
99     if not poll.open:
100         return ['Poll is now closed.', poll.votes]
101     if player in poll.voters:
102         return ['You have already voted in this poll.', poll.votes]
103     return ['You have not voted in this poll yet.']
104
105 def make_new_poll_command(instance, player, arguments):
106     """ Make a new poll.
107
108     Args:
109         instance: The game instance to add the poll to.

```



```

110     player: The email of the player creating the poll.
111     arguments: A two item list containing the question and a
112     second list of 2-5 options.
113
114     Returns:
115     Returns a list with information about the poll just created.
116     See get_poll_return_list for its format.
117
118     Raises:
119     ValueError if the player is not in the instance.
120     """
121     instance.check_player(player)
122     if not arguments[0]:
123         raise ValueError('Question cannot be empty')
124     size = len(arguments[1])
125     if size < 2 or size > 5:
126         raise ValueError('Incorrect number of options for poll. ' +
127                           'Must be between two and five.')
128
129     poll = Message(parent = instance, sender = player,
130                   msg_type = 'poll', recipient = '')
131     poll.put()
132     arguments.append(poll.key().id())
133     poll.content = simplejson.dumps(arguments)
134     poll.votes = [0] * size
135     poll.open = True
136     poll.voters = ['']
137     poll.put()
138     return get_poll_return_list(poll)
139
140 def close_poll_command(instance, player, arguments):
141     """ Close an existing poll.
142
143     Args:
144     instance: The parent GameInstance model of the poll.
145     player: The email of the player closing the poll. Must be the
146     poll's creator.
147     arguments: A one argument list with the poll's id number.
148
149     Returns:
150     A list with information about the poll just closed. See
151     get_poll_return_list for its format.
152
153     Raises:
154     ValueError if player is not the creator of the poll.
155     ValueError if the player is not in the instance.
156     """
157     instance.check_player(player)
158     poll = get_poll(instance, arguments[0])
159     if poll.sender != player:
160         raise ValueError('Only the person that created this poll may close
161                           it.')
162     poll.open = False
163     poll.msg_type = 'closed_poll'

```

```

163 poll.put()
164 return get_poll_return_list(poll)
165
166 def delete_poll_command(instance, player, arguments):
167     """ Delete an existing poll.
168
169     Args:
170         instance: The parent GameInstance model of the poll.
171         player: The email of the player closing the poll. Must be the
172             poll's creator.
173         arguments: A one argument list with the poll's id number.
174
175     Returns:
176         True if the deletion is successful.
177
178     Raises:
179         ValueError if player is not the creator of the poll.
180         ValueError if the player is not in the instance.
181     """
182     instance.check_player(player)
183     poll = get_poll(instance, arguments[0])
184     if poll.sender != player:
185         raise ValueError('Only the person that created this poll may
186             delete it.')
187     db.delete(poll)
188     return [True]
189
190 def get_poll_info_command(instance, player, arguments):
191     """ Get information about an existing poll.
192
193     Args:
194         instance: The parent GameInstance model of the poll.
195         player: The email of the player requesting information. Must
196             be the poll's creator.
197         arguments: A one argument list with the poll's id number.
198
199     Returns:
200         A list with information about the poll. See
201         get_poll_return_list for its format.
202
203     Raises:
204         ValueError if player is not the creator of the poll.
205         ValueError if the player is not in the instance.
206     """
207     instance.check_player(player)
208     poll = get_poll(instance, arguments[0])
209     if poll.sender != player:
210         raise ValueError('Only the person that created the poll can'
211             + 'request its information.')
212     return get_poll_return_list(poll)
213
214 def get_my_polls_command(instance, player, arguments = None):
215     """ Get the polls created by a player in the instance.

```

```

216
217     Args:
218         instance: The parent GameInstance model of the polls.
219         player: The email of the player requesting the polls.
220         arguments: Not used, can be any value.
221
222     Finds all polls created by this player.
223
224     Returns:
225         A list of two item lists with each containing the
226         id number of the poll and its question.
227
228     Raises:
229         ValueError if the player is not in the instance.
230     """
231     instance.check_player(player)
232     query = instance.get_messages_query('', '', sender = player)
233     polls = query.fetch(1000)
234     return [[poll.key().id(), poll.get_content()[0]] for poll in polls[:
235         :-1]]
236
237 def get_poll(instance, argument):
238     """ Get a poll database model.
239
240     Args:
241         instance: The parent GameInstance database model of the poll.
242         argument: The poll id argument from the server command
243         arguments list.
244
245     Returns:
246         A Message database model of the poll.
247
248     Raises:
249         ValueError if argument fails to parse to an int or the
250         poll doesn't exist in the database.
251     """
252     try:
253         poll_id = int(argument)
254     except ValueError:
255         raise ValueError('Poll id failed to parse to a number.')
256
257     poll_key = db.Key.from_path('Message', poll_id,
258                                parent = instance.key())
259     poll = db.get(poll_key)
260
261     if poll is None:
262         raise ValueError('Poll no longer exists.')
263     return poll
264
265 def get_poll_return_list(poll):
266     """ Get a list to return to the GameClient component for a poll.
267
268     Args:
269         poll: A Message database model that is a poll.

```

```
269
270     Returns:
271         A list with the following five items:
272             The poll question.
273             The poll options as a list.
274             The poll id number.
275             The poll votes as a list.
276             Whether the poll is open.
277     """
278     content = poll.get_content()
279     content.extend([poll.votes, poll.open])
280     return content
```

Appendix B

Game Client Code

This appendix includes selected JAVA source files from the App Inventor component runtime. Source code in this appendix is not presented in its directory structure as in Appendix A.

B.1 GameClient Component

B.1: GameClient.java - The GameClient component.

```
1  // Copyright 2009 Google Inc. All Rights Reserved.
2
3  package com.google.devtools.simple.runtime.components.android;
4
5  import com.google.devtools.simple.common.ComponentCategory;
6  import com.google.devtools.simple.runtime.annotations.
    DesignerComponent;
7  import com.google.devtools.simple.runtime.annotations.DesignerProperty
    ;
8  import com.google.devtools.simple.runtime.annotations.SimpleEvent;
9  import com.google.devtools.simple.runtime.annotations.SimpleFunction;
10 import com.google.devtools.simple.runtime.annotations.SimpleObject;
11 import com.google.devtools.simple.runtime.annotations.SimpleProperty;
12 import com.google.devtools.simple.runtime.annotations.UsesPermissions;
13 import com.google.devtools.simple.runtime.components.android.collect.
    Lists;
14 import com.google.devtools.simple.runtime.components.android.util.
    AsyncCallbackPair;
15 import com.google.devtools.simple.runtime.components.android.util.
    AsyncUtil;
16 import com.google.devtools.simple.runtime.components.android.util.
    GameInstance;
```

```

17 import com.google.devtools.simple.runtime.components.android.util.
    LoginServiceUtil;
18 import com.google.devtools.simple.runtime.components.android.util.
    PlayerListDelta;
19 import com.google.devtools.simple.runtime.components.android.util.
    WebServiceUtil;
20 import com.google.devtools.simple.runtime.components.util.JsonUtil;
21 import com.google.devtools.simple.runtime.components.util.YailList;
22 import com.google.devtools.simple.runtime.errors.YailRuntimeError;
23 import com.google.devtools.simple.runtime.events.EventDispatcher;
24
25 import android.app.Activity;
26 import android.os.Handler;
27 import android.util.Log;
28
29 import org.apache.http.NameValuePair;
30 import org.apache.http.message.BasicNameValuePair;
31 import org.json.JSONArray;
32 import org.json.JSONException;
33 import org.json.JSONObject;
34
35 import java.util.ArrayList;
36 import java.util.List;
37
38 /**
39  * GameClient provides a way for AppInventor applications to
40  * communicate with online game servers. This allows users to create
41  * games that are coordinated and managed in the cloud.
42  *
43  * Most communication is done by sending keyed messages back and
44  * forth between the client and the server in the form of YailLists.
45  * The server and game client can then switch on the keys and perform
46  * more complex operations on the data. In addition, game servers can
47  * implement a library of server commands that can perform complex
48  * functions on the server and send back responses that are converted
49  * into YailLists and sent back to the component. For more
50  * information about server commands, consult the game server code
51  * at http://code.google.com/p/app-inventor-for-android/
52  *
53  * Games instances are uniquely determined by a game id and an
54  * instance id. In general, each App Inventor program should have
55  * its own game id. Then, when running different instances of that
56  * program, new instance ids should be used. Players are
57  * represented uniquely by the email address registered to their
58  * phones.
59  *
60  * All call functions perform POSTs to a web server. Upon successful
61  * completion of these POST requests, FunctionCompleted will be
62  * triggered with the function name as an argument. If the post
63  * fails, WebServiceError will trigger with the function name and the
64  * error message as arguments. These calls allow for application
65  * creators to deal with web service failures and keep track of the
66  * success or failure of these operations. The only exception to this
67  * is when the return value from the server has the incorrect game id

```

```

68  * or instance id. In this case, the response is completely ignored
69  * and neither of these events will trigger.
70  *
71  * @author billmag@google.com (Bill Magnuson)
72  *
73  */
74  @DesignerComponent(
75      description = "Provides a way for applications to communicate with
        online game servers",
76      category = ComponentCategory.EXPERIMENTAL)
77  @SimpleObject
78  @UsesPermissions(
79      permissionNames = "android.permission.INTERNET, com.google.android
        .googleapps.permission.GOOGLE_AUTH")
80  public class GameClient implements OnResumeListener, OnStopListener {
81
82      private static final String LOG_TAG = "GameClient";
83
84      // Parameter keys
85      private static final String GAME_ID_KEY = "gid";
86      private static final String INSTANCE_ID_KEY = "iid";
87      private static final String PLAYER_ID_KEY = "pid";
88      private static final String INVITEE_KEY = "inv";
89      private static final String LEADER_KEY = "leader";
90      private static final String COUNT_KEY = "count";
91      private static final String TYPE_KEY = "type";
92      private static final String INSTANCE_PUBLIC_KEY = "makepublic";
93      private static final String MESSAGE_RECIPIENTS_KEY = "mrec";
94      private static final String MESSAGE_CONTENT_KEY = "contents";
95      private static final String MESSAGE_TIME_KEY = "mtime";
96      private static final String MESSAGE_SENDER_KEY = "msender";
97      private static final String COMMAND_TYPE_KEY = "command";
98      private static final String COMMAND_ARGUMENTS_KEY = "args";
99      private static final String SERVER_RETURN_VALUE_KEY = "response";
100     private static final String MESSAGES_LIST_KEY = "messages";
101     private static final String ERROR_RESPONSE_KEY = "e";
102     private static final String PUBLIC_LIST_KEY = "public";
103     private static final String JOINED_LIST_KEY = "joined";
104     private static final String INVITED_LIST_KEY = "invited";
105     private static final String PLAYERS_LIST_KEY = "players";
106
107     // Command keys
108     private static final String GET_INSTANCE_LISTS_COMMAND = "
        getinstancelists";
109     private static final String GET_MESSAGES_COMMAND = "messages";
110     private static final String INVITE_COMMAND = "invite";
111     private static final String JOIN_INSTANCE_COMMAND = "joininstance";
112     private static final String LEAVE_INSTANCE_COMMAND = "leaveinstance"
        ;
113     private static final String NEW_INSTANCE_COMMAND = "newinstance";
114     private static final String NEW_MESSAGE_COMMAND = "newmessage";
115     private static final String SERVER_COMMAND = "servercommand";
116     private static final String SET_LEADER_COMMAND = "setleader";
117

```

```

118  // URL for accessing the game server
119  private String serviceUrl;
120  private String gameId;
121  private GameInstance instance;
122  private Handler androidUIHandler;
123  private Activity activityContext;
124
125  private String userEmailAddress = "";
126
127  // Game instances in the current GameId that this player has joined
128  private List<String> joinedInstances;
129  // Game instances to which this player has been invited
130  private List<String> invitedInstances;
131  // Game instances which have been made public.
132  private List<String> publicInstances;
133
134  /**
135   * Creates a new GameClient component.
136   *
137   * @param container the Form that this component is contained in.
138   */
139  public GameClient(ComponentContainer container) {
140      // Note that although this is creating a new Handler there is
141      // only one UI thread in an Android app and posting to this
142      // handler queues up a Runnable for execution on that thread.
143      androidUIHandler = new Handler();
144      activityContext = container.$context();
145      Form form = container.$form();
146      form.registerForOnResume(this);
147      form.registerForOnStop(this);
148      gameId = "";
149      instance = new GameInstance("");
150      joinedInstances = Lists.newArrayList();
151      invitedInstances = Lists.newArrayList();
152      publicInstances = Lists.newArrayList();
153      serviceUrl = "http://appinvgameserver.appspot.com";
154
155      // This needs to be done in a separate thread since it uses
156      // a blocking service to complete and will cause the UI to hang
157      // if it happens in the constructor.
158      AsyncUtil.runAsynchronously(new Runnable() {
159          @Override
160          public void run() {
161              userEmailAddress = LoginServiceUtil.getPhoneEmailAddress(
162                  activityContext);
163              if (!userEmailAddress.equals("")) {
164                  UserEmailAddressSet(userEmailAddress);
165              }
166          }
167      });
168
169  }
170  //-----

```



```

171 // Properties
172
173 /**
174  * Returns a string indicating the game name for this application.
175  * The same game ID can have one or more game instances.
176  */
177 @SimpleProperty
178 public String GameId() {
179     return gameId;
180 }
181
182 /**
183  * Specifies a string indicating the family of the current game
184  * instance. The same game ID can have one or more game instance
185  * IDs.
186  */
187 // Only exposed in the designer to enforce that each GameClient
188 // instance should be made for a single GameId.
189 @DesignerProperty(
190     editorType = DesignerProperty.PROPERTY_TYPE_STRING,
191     defaultValue = "\\\"\\\"")
192 public void GameId(String id) {
193     this.gameId = id;
194 }
195
196 /**
197  * Returns the game instance id. Taken together, the game ID and
198  * the instance ID uniquely identify the game.
199  */
200 @SimpleProperty
201 public String InstanceId() {
202     return instance.getInstanceId();
203 }
204
205 /**
206  * Returns the set of game instances to which this player has been
207  * invited but has not yet joined. To ensure current values are
208  * returned, first invoke {@link #GetInstanceLists}.
209  */
210 @SimpleProperty
211 public List<String> InvitedInstances() {
212     return invitedInstances;
213 }
214
215 /**
216  * Returns the set of game instances in which this player is
217  * participating. To ensure current values are returned, first
218  * invoke {@link #GetInstanceLists}.
219  */
220 @SimpleProperty
221 public List<String> JoinedInstances() {
222     return joinedInstances;
223 }
224

```

```

225  /**
226   * Returns the game's leader. At any time, each game instance has
227   * only one leader, but the leader may change with time.
228   * Initially, the leader is the game instance creator. Application
229   * writers determine special properties of the leader. The leader
230   * value is updated each time a successful communication is made
231   * with the server.
232   */
233  @SimpleProperty
234  public String Leader() {
235      return instance.getLeader();
236  }
237
238  /**
239   * Returns the current set of players for this game instance. Each
240   * player is designated by an email address, which is a string. The
241   * list of players is updated each time a successful communication
242   * is made with the game server.
243   */
244  @SimpleProperty
245  public List<String> Players() {
246      return instance.getPlayers();
247  }
248
249  /**
250   * Returns the set of game instances that have been marked public.
251   * To ensure current values are returned, first
252   * invoke {@link #GetInstanceLists}.
253   */
254  @SimpleProperty
255  public List<String> PublicInstances() {
256      return publicInstances;
257  }
258
259  /**
260   * The URL of the game server.
261   *
262   */
263  @SimpleProperty
264  public String ServiceUrl() {
265      return serviceUrl;
266  }
267
268  /**
269   * Set the URL of the game server.
270   *
271   * @param url The URL (include initial http://).
272   */
273  @DesignerProperty(
274      editorType = DesignerProperty.PROPERTY_TYPE_STRING,
275      defaultValue = "\"http://appinvgameserver.appspot.com\""
276  )
277  public void ServiceURL(String url) {
278      if (url.endsWith("/")) {
279          this.serviceUrl = url.substring(0, url.length() - 1);

```

```

279     } else {
280         this.serviceUrl = url;
281     }
282 }
283
284 /**
285  * Returns the registered email address that is being used as the
286  * player id for this game client.
287  */
288 @SimpleProperty
289 public String UserEmailAddress() {
290     if (userEmailAddress.equals("")) {
291         Info("User email address is empty.");
292     }
293     return userEmailAddress;
294 }
295
296 /**
297  * Changes the player of this game by changing the email address
298  * used to communicate with the server.
299  *
300  * This should only be used during development. Games should not
301  * allow players to set their own email address.
302  *
303  * @param emailAddress The email address to set the current player
304  * id to.
305  */
306 @SimpleProperty
307 public void UserEmailAddress(String emailAddress) {
308     userEmailAddress = emailAddress;
309     UserEmailAddressSet(emailAddress);
310 }
311
312 //-----
313 // Event Handlers
314
315 /**
316  * Indicates that a server request from a function call has
317  * completed. This can be used to control a polling loop or
318  * otherwise respond to server request completions.
319  *
320  * @param functionName The name of the App Inventor function that
321  * finished.
322  */
323 @SimpleEvent(description = "Indicates that a function call completed
324 .")
325 public void FunctionCompleted(final String functionName) {
326     androidUIHandler.post(new Runnable() {
327         public void run() {
328             Log.d(LOG_TAG, "Request completed: " + functionName);
329             EventDispatcher.dispatchEvent(GameClient.this, "
330 FunctionCompleted", functionName);
331         }
332     });
333 }

```

```

331
332  /**
333   * Default Initialize event handler. Ensures that the gameId was
334   * set by the game creator.
335   */
336  @SimpleEvent
337  public void Initialize() {
338      Log.d(LOG_TAG, "Initialize");
339      if (gameId.equals("")) {
340          throw new YailRuntimeError("Game Id must not be empty.", "
GameClient Configuration Error.");
341      }
342      EventDispatcher.dispatchEvent(this, "Initialize");
343  }
344
345  /**
346   * Indicates that a GetMessages call received a message. This could
347   * be invoked multiple times for a single call to GetMessages.
348   *
349   * @param type The type of the message received.
350   * @param contents The message's contents. Consists of a list
351   * nested to arbitrary depth that includes string, boolean and
352   * number values.
353   */
354  @SimpleEvent(description = "Indicates that a new message has " +
355      "been received.")
356  public void GotMessage(final String type, final String sender, final
List<Object> contents) {
357      Log.d(LOG_TAG, "Got message of type " + type);
358      androidUIHandler.post(new Runnable() {
359          public void run() {
360              EventDispatcher.dispatchEvent(GameClient.this, "GotMessage",
type, sender, contents);
361          }
362      });
363
364  /**
365   * Indicates that InstanceId has changed due to the creation of a
366   * new instance or setting the InstanceId.
367   *
368   * @param instanceId The id of the instance the player is now in.
369   */
370  @SimpleEvent(description = "Indicates that the InstanceId " +
371      "property has hanged as a result of calling " +
372      "MakeNewInstance or SetInstanceId.")
373  public void InstanceIdChanged(final String instanceId) {
374      Log.d(LOG_TAG, "Instance id changed to " + instanceId);
375      androidUIHandler.post(new Runnable() {
376          public void run() {
377              EventDispatcher.dispatchEvent(GameClient.this, "
InstanceIdChanged", instanceId);
378          }
379      });
380

```

```

381  /**
382   * Indicates a user has been invited to this game instance by
383   * another player.
384   *
385   * @param instanceId The id of the new game instance.
386   */
387  @SimpleEvent(
388      description = "Indicates that a user has been invited to " +
389          "this game instance.")
390  public void Invited(final String instanceId) {
391      Log.d(LOG_TAG, "Player invited to " + instanceId);
392      androidUIHandler.post(new Runnable() {
393          public void run() {
394              EventDispatcher.dispatchEvent(GameClient.this, "Invited",
395                  instanceId);
396          }
397      });
398  }
399  /**
400   * Indicates this game instance has a new leader. This could happen
401   * in response to a call to SetLeader or by the side effects of a
402   * server command performed by any player in the game.
403   *
404   * Since the current leader is sent back with every server
405   * response, NewLeader can trigger after making any server call.
406   *
407   * @param playerId The email address of the new leader.
408   */
409  @SimpleEvent(description = "Indicates that this game has a new " +
410      "leader as specified through SetLeader")
411  public void NewLeader(final String playerId) {
412      androidUIHandler.post(new Runnable() {
413          public void run() {
414              Log.d(LOG_TAG, "Leader change to " + playerId);
415              EventDispatcher.dispatchEvent(GameClient.this, "NewLeader",
416                  playerId);
417          }
418      });
419  }
420  /**
421   * Indicates this game instance was created as specified via
422   * MakeNewInstance. The creating player is automatically the leader
423   * of the instance and the InstanceId property has already been set
424   * to this new instance.
425   *
426   * @param instanceId The id of the newly created game instance.
427   */
428  @SimpleEvent(description = "Indicates that a new instance was " +
429      "successfully created after calling MakeNewInstance.")
430  public void NewInstanceMade(final String instanceId) {
431      androidUIHandler.post(new Runnable() {
432          public void run() {
433              Log.d(LOG_TAG, "New instance made: " + instanceId);

```

```

432         EventDispatcher.dispatchEvent(GameClient.this, "
    NewInstanceMade", instanceId);
433     });
434 }
435
436 /**
437  * Indicates that a player has joined this game instance.
438  *
439  * @param playerId The email address of the new player.
440  */
441 @SimpleEvent(description = "Indicates that a new player has " +
442     "joined this game instance.")
443 public void PlayerJoined(final String playerId) {
444     androidUIHandler.post(new Runnable() {
445         public void run() {
446             if (!playerId.equals(UserEmailAddress())) {
447                 Log.d(LOG_TAG, "Player joined: " + playerId);
448                 EventDispatcher.dispatchEvent(GameClient.this, "PlayerJoined
    ", playerId);
449             }
450         }
451     });
452 }
453
454 /**
455  * Indicates that a player has left this game instance.
456  *
457  * @param playerId The email address of the player that left.
458  */
459 @SimpleEvent(description = "Indicates that a player has left " +
460     "this game instance.")
461 public void PlayerLeft(final String playerId) {
462     androidUIHandler.post(new Runnable() {
463         public void run() {
464             Log.d(LOG_TAG, "Player left: " + playerId);
465             EventDispatcher.dispatchEvent(GameClient.this, "PlayerLeft",
    playerId);
466         }
467     });
468 }
469
470 /**
471  * Indicates that an attempt to complete a server command failed on
472  * the server.
473  *
474  * @param command The command requested.
475  * @param arguments The arguments sent to the command.
476  */
477 @SimpleEvent(
478     description = "Indicates that a server command failed.")
479 public void ServerCommandFailure(final String command, final
    Yaillist arguments) {
480     androidUIHandler.post(new Runnable() {
481         public void run() {
482             Log.d(LOG_TAG, "Server command failed: " + command);
483             EventDispatcher.dispatchEvent(GameClient.this, "
    ServerCommandFailure", command, arguments);

```

```

481         }));
482     }
483
484     /**
485      * Indicates that a ServerCommand completed.
486      *
487      * @param command The key for the command that resulted in this
488      * response.
489      * @param response The server response. This consists of a list
490      * nested to arbitrary depth that includes string, boolean and
491      * number values.
492      */
493     @SimpleEvent(description = "Indicates that a server command " +
494         "returned successfully.")
495     public void ServerCommandSuccess(final String command, final List<
496         Object> response) {
497         Log.d(LOG_TAG, command + " server command returned.");
498         androidUIHandler.post(new Runnable() {
499             public void run() {
500                 EventDispatcher.dispatchEvent(GameClient.this,
501                     "ServerCommandSuccess", command, response);
502             }
503         });
504
505     /**
506      * Indicates that the user email address property has been
507      * successfully set. This event should be used to initialize
508      * any web service functions.
509      *
510      * This separate event was required because the email address was
511      * unable to be first fetched from the the UI thread without
512      * causing programs to hang. GameClient will now start fetching
513      * the user email address in its constructor and trigger this event
514      * when it finishes.
515      */
516     @SimpleEvent(description = "Indicates that the user email " +
517         "address has been set.")
518     public void UserEmailAddressSet(final String emailAddress) {
519         Log.d(LOG_TAG, "Email address set.");
520         androidUIHandler.post(new Runnable() {
521             public void run() {
522                 EventDispatcher.dispatchEvent(GameClient.this, "
523                     UserEmailAddressSet", emailAddress);
524             }
525         });
526
527     //-----
528     // Message events
529
530     /**
531      * Indicates that something has occurred which the player should be
532      * somehow informed of.
533      *
534      * @param message the message.

```

```

533     */
534     @SimpleEvent(description = "Indicates that something has " +
535         "occurred which the player should know about.")
536     public void Info(final String message) {
537         Log.d(LOG_TAG, "Info: " + message);
538         androidUIHandler.post(new Runnable() {
539             public void run() {
540                 EventDispatcher.dispatchEvent(GameClient.this, "Info", message
541             );
542             });
543     }
544
545     /**
546      * Indicates that the attempt to communicate with the web service
547      * resulted in an error.
548      *
549      * @functionName The name of the function call that caused this
550      * error.
551      * @param message the error message
552     */
553     @SimpleEvent(description = "Indicates that an error occurred " +
554         "while communicating with the web server.")
555     public void WebServiceError(final String functionName, final String
556         message) {
557         Log.e(LOG_TAG, "WebServiceError: " + message);
558         androidUIHandler.post(new Runnable() {
559             public void run() {
560                 EventDispatcher.dispatchEvent(GameClient.this, "
561                 WebServiceError", functionName, message);
562             });
563     }
564
565     //-----
566     // Functions
567
568     /**
569      * Updates the current InstancesJoined and InstancesInvited lists.
570      *
571      * If the player has been invited to new instances an Invited
572      * event will be raised for each new instance.
573     */
574     @SimpleFunction(description = "Updates the InstancesJoined and " +
575         "InstancesInvited lists. This procedure can be called " +
576         "before setting the InstanceId.")
577     public void GetInstanceLists() {
578         AsyncUtil.runAsynchronously(new Runnable() {
579             public void run() { postGetInstanceLists(); });
580     }
581
582     private void postGetInstanceLists() {
583         AsyncCallbackPair<JSONObject> readMessagesCallback = new
584         AsyncCallbackPair<JSONObject>() {
585             public void onSuccess(final JSONObject response) {

```



```

583         processInstanceLists(response);
584         FunctionCompleted("GetInstanceLists");
585     }
586     public void onFailure(final String message) {
587         WebServiceError("GetInstanceLists", "Failed to get up to date
instance lists.");
588     }
589 };
590
591 postCommandToGameServer(GET_INSTANCE_LISTS_COMMAND,
592     Lists.<NameValuePair>newArrayList(
593         new BasicNameValuePair(GAME_ID_KEY, GameId()),
594         new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
595         new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress())
596     ),
597     readMessagesCallback);
598
599 private void processInstanceLists(JSONObject instanceLists){
600     try {
601         joinedInstances = JsonUtil.getStringListFromJsonArray(
instanceLists.
602             getJSONArray(JOINED_LIST_KEY));
603
604         publicInstances = JsonUtil.getStringListFromJsonArray(
instanceLists.
605             getJSONArray(PUBLIC_LIST_KEY));
606
607         List<String> receivedInstancesInvited = JsonUtil.
getStringListFromJsonArray(instanceLists.
608             getJSONArray(INVITED_LIST_KEY));
609
610         if (!receivedInstancesInvited.equals(invitedInstances())) {
611             List<String> oldList = invitedInstances;
612             invitedInstances = receivedInstancesInvited;
613             List<String> newInvites = new ArrayList<String>(
receivedInstancesInvited);
614             newInvites.removeAll(oldList);
615
616             for (final String instanceInvited : newInvites) {
617                 Invited(instanceInvited);
618             }
619         }
620
621     } catch (JSONException e) {
622         Log.w(LOG_TAG, e);
623         Info("Instance lists failed to parse.");
624     }
625 }
626
627 /**
628  * Retrieves messages of the specified type.
629  *
630  * Requests that only messages which have not been seen during

```

```

631      * the current session are returned. Messages will be processed
632      * in chronological order with the oldest first, however, only
633      * the count newest messages will be retrieved. This means that
634      * one could "miss out" on some messages if they request less than
635      * the number of messages created since the last request for
636      * that message type.
637      *
638      * Setting type to the empty string will fetch all message types.
639      * Even though those message types were not specifically requested,
640      * their most recent message time will be updated. This keeps
641      * players from receiving the same message again if they later
642      * request the specific message type.
643      *
644      * Note that the message receive times are not updated until after
645      * the messages are actually received. Thus, if multiple message
646      * requests are made before the previous ones return, they could
647      * send stale time values and thus receive the same messages more
648      * than once. To avoid this, application creators should wait for
649      * the get messages function to return before calling it again.
650      *
651      * @param type The type of message to retrieve. If the empty string
652      * is used as the message type then all message types will be
653      * requested.
654      * @param count The maximum number of messages to retrieve. This
655      * should be an integer from 1 to 1000.
656      */
657      @SimpleFunction(
658          description = "Retrieves messages of the specified type.")
659      public void GetMessages(final String type, final int count) {
660          AsyncUtil.runAsynchronously(new Runnable() {
661              public void run() { postGetMessages(type, count); });
662      }
663
664      private void postGetMessages(final String requestedType, final int
        count) {
665          AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<
        JSONObject>() {
666              public void onSuccess(final JSONObject result) {
667                  try {
668                      int count = result.getInt(COUNT_KEY);
669                      JSONArray messages = result.getJSONArray(MESSAGES_LIST_KEY);
670                      for (int i = 0; i < count; i++) {
671                          JSONObject message = messages.getJSONObject(i);
672                          String type = message.getString(TYPE_KEY);
673                          String sender = message.getString(MESSAGE_SENDER_KEY);
674                          String time = message.getString(MESSAGE_TIME_KEY);
675                          List<Object> contents = JsonUtil.getListFromJsonArray(
        message.
676                              getJSONArray(MESSAGE_CONTENT_KEY));
677                          // Assumes that the server is going to return messages in
678                          // chronological order.
679                          if (requestedType.equals("")) {
680                              instance.putMessageTime(requestedType, time);
681                          }

```

```

682         instance.putMessageTime(type, time);
683         GotMessage(type, sender, contents);
684     }
685     } catch (JSONException e) {
686         Log.w(LOG_TAG, e);
687         Info("Failed to parse messages response.");
688     }
689     FunctionCompleted("GetMessages");
690 }
691
692 public void onFailure(String message) {
693     WebServiceError("GetMessages", message);
694 }
695 };
696
697 if (InstanceId().equals("")) {
698     Info("You must join an instance before attempting to fetch
699 messages.");
700     return;
701 }
702
703 postCommandToGameServer(GET_MESSAGES_COMMAND,
704     Lists.<NameValuePair>newArrayList(
705         new BasicNameValuePair(GAME_ID_KEY, gameId()),
706         new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
707         new BasicNameValuePair(PLAYER_ID_KEY, userEmailAddress()),
708         new BasicNameValuePair(COUNT_KEY, new Integer(count).
709 toString()),
710         new BasicNameValuePair(MESSAGE_TIME_KEY, instance.
711 getMessageTime(requestedType)),
712         new BasicNameValuePair(TYPE_KEY, requestedType)),
713     myCallback);
714 }
715
716 /**
717  * Invites a player to this game instance.
718  *
719  * Players implicitly accept invitations when they join games by
720  * setting the instance id in their GameClient.
721  *
722  * Invitations remain active as long as the game instance exists.
723  *
724  * @param playerEmail a string containing the email address of the
725  * player to become leader. The email should be in one of the
726  * following formats:<br>"Name O. Person
727  * &ltname.o.person@gmail.com&gt;"<br>"name.o.person@gmail.com".
728  */
729 @SimpleFunction(
730     description = "Invites a player to this game instance.")
731 public void Invite(final String playerEmail) {
732     AsyncUtil.runAsynchronously(new Runnable() {
733         public void run() { postInvite(playerEmail); }});
734 }

```

```

733 private void postInvite(final String inviteeEmail) {
734     AsyncCallbackPair<JSONObject> inviteCallback = new
        AsyncCallbackPair<JSONObject>() {
735         public void onSuccess(final JSONObject response) {
736             try {
737                 String invitedPlayer = response.getString(INVITEE_KEY);
738
739                 if (invitedPlayer.equals("")) {
740                     Info(invitedPlayer + " was already invited.");
741                 } else {
742                     Info("Successfully invited " + invitedPlayer + ".");
743                 }
744             } catch (JSONException e) {
745                 Log.w(LOG_TAG, e);
746                 Info("Failed to parse invite player response.");
747             }
748             FunctionCompleted("Invite");
749         }
750         public void onFailure(final String message) {
751             WebServiceError("Invite", message);
752         }
753     };
754
755     if (InstanceId().equals("")) {
756         Info("You must have joined an instance before you can invite new
        players.");
757         return;
758     }
759
760     postCommandToGameServer(INVITE_COMMAND,
761         Lists.<NameValuePair>newArrayList(
762             new BasicNameValuePair(GAME_ID_KEY, GameId()),
763             new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
764             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
765             new BasicNameValuePair(INVITEE_KEY, inviteeEmail)),
766         inviteCallback);
767 }
768
769 /**
770  * Requests to leave the current instance. If the player is the
771  * current leader, the lead will be passed to another player.
772  *
773  * If there are no other players left in the instance after the
774  * current player leaves, the instance will become unjoinable.
775  *
776  * Upon successful completion of this command, the instance
777  * lists will be updated and InstanceId will be set back to the
778  * empty string.
779  *
780  * Note that while this call does clear the leader and player
781  * lists, no NewLeader or PlayerLeft events are raised.
782  */
783 @SimpleFunction(description = "Leaves the current instance.")
784 public void LeaveInstance() {

```

```

785     AsyncUtil.runAsynchronously(new Runnable() {
786         public void run() {
787             postLeaveInstance();
788         }
789     });
790 }
791
792 private void postLeaveInstance() {
793     AsyncCallbackPair<JSONObject> setInstanceCallback = new
794     AsyncCallbackPair<JSONObject>() {
795         public void onSuccess(final JSONObject response) {
796             SetInstance("");
797             processInstanceLists(response);
798             FunctionCompleted("LeaveInstance");
799         }
800         public void onFailure(final String message) {
801             WebServiceError("LeaveInstance", message);
802         }
803     };
804
805     postCommandToGameServer(LEAVE_INSTANCE_COMMAND,
806         Lists.<NameValuePair>newArrayList(
807             new BasicNameValuePair(GAME_ID_KEY, GameId()),
808             new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
809             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress())
810         ),
811         setInstanceCallback);
812 }
813
814 /**
815  * Creates a new game instance. The instance has a unique
816  * instanceId, and the leader is the player who created it. The
817  * player that creates the game automatically joins it without
818  * being sent an invitation.
819  *
820  * The actual instance id could differ from the instanceId
821  * specified because the game server will enforce uniqueness. The
822  * actual instanceId will be provided to AppInventor when a
823  * NewInstanceMade event triggers upon successful completion of
824  * this server request.
825  *
826  * @param instanceId A string to use as for the instance
827  * id. If no other instance exists with this id, the new instance
828  * will have this id. However, since the id must be unique, if
829  * another instance exists with the same one, then a number
830  * will be appended to the end of this prefix.
831  * @param makePublic A boolean indicating whether or not the
832  * instance should be publicly viewable and able to be joined by
833  * anyone.
834  */
835 @SimpleFunction(description = "Asks the server to create a new " +
    "instance of this game.")
    public void MakeNewInstance(final String instanceId, final boolean
        makePublic) {

```

```

836     AsyncUtil.runAsynchronously(new Runnable() {
837         public void run() { postMakeNewInstance(instanceId, makePublic);
            });
838     }
839
840     private void postMakeNewInstance(final String requestedInstanceId,
841         final Boolean makePublic) {
842         AsyncCallbackPair<JSONObject> makeNewGameCallback = new
843         AsyncCallbackPair<JSONObject>() {
844             public void onSuccess(final JSONObject response) {
845                 processInstanceLists(response);
846                 NewInstanceMade(instanceId());
847                 FunctionCompleted("MakeNewInstance");
848             }
849             public void onFailure(final String message) {
850                 WebServiceError("MakeNewInstance", message);
851             }
852         };
853
854         postCommandToGameServer(NEW_INSTANCE_COMMAND,
855             Lists.<NameValuePair>newArrayList(
856                 new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
857                 new BasicNameValuePair(GAME_ID_KEY, GameId()),
858                 new BasicNameValuePair(INSTANCE_ID_KEY,
859                     requestedInstanceId),
860                 new BasicNameValuePair(INSTANCE_PUBLIC_KEY, makePublic.
861                     toString())),
862             makeNewGameCallback, true);
863     }
864
865     /**
866     * Creates a new message and sends it to the stated recipients.
867     *
868     * @param type A "key" for the message. This identifies the type of
869     * message so that when other players receive the message they know
870     * how to properly handle it.
871     * @param recipients If set to an empty list, the server will send
872     * this message with a blank set of recipients, meaning that all
873     * players in the instance are able to retrieve it. To limit the
874     * message receipt to a single person or a group of people,
875     * recipients should be a list of the email addresses of the people
876     * meant to receive the message. Each email should be in one of the
877     * following formats:<br>
878     * "Name O. Person &lt;name.o.person@gmail.com>"<br>
879     * "name.o.person@gmail.com"
880     * @param contents the contents of the message. This can be any
881     * AppInventor data value.
882     */
883     @SimpleFunction(description = "Sends a keyed message to all " +
884         "recipients in the recipients list. The message will " +
885         "consist of the contents list.")
886     public void SendMessage(final String type, final YailList recipients
887         , final YailList contents) {
888         AsyncUtil.runAsynchronously(new Runnable() {

```

```

884     public void run() { postNewMessage(type, recipients, contents);
885     });
886 }
887 private void postNewMessage(final String type, YailList recipients,
888     YailList contents){
889     AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<
890     JSONObject>(){
889         public void onSuccess(final JSONObject response) {
890             FunctionCompleted("SendMessage");
891         }
892         public void onFailure(final String message) {
893             WebServiceError("SendMessage", message);
894         }
895     };
896
897     if (InstanceId().equals("")) {
898         Info("You must have joined an instance before you can send
899         messages.");
900         return;
901     }
902     postCommandToGameServer(NEW_MESSAGE_COMMAND,
903         Lists.<NameValuePair>newArrayList(
904             new BasicNameValuePair(GAME_ID_KEY, GameId()),
905             new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
906             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
907             new BasicNameValuePair(TYPE_KEY, type),
908             new BasicNameValuePair(MESSAGE_RECIPIENTS_KEY, recipients.
909             toJSONString()),
910             new BasicNameValuePair(MESSAGE_CONTENT_KEY, contents.
911             toJSONString()),
912             new BasicNameValuePair(MESSAGE_TIME_KEY, instance.
913             getMessageTime(type))),
914             myCallback);
915 }
916
917 /**
918  * Submits a command to the game server. Server commands are
919  * custom actions that are performed on the server. The arguments
920  * required and return value of a server command depend on its
921  * implementation.
922  *
923  * For more information about server commands, consult the game
924  * server code at:
925  * http://code.google.com/p/app-inventor-for-android/
926  *
927  * @param command The name of the server command.
928  * @param arguments The arguments to pass to the server to specify
929  * how to execute the command.
930  */
931 @SimpleFunction(description = "Sends the specified command to " +
932     "the game server.")

```

```

930 public void ServerCommand(final String command, final YaillList
    arguments) {
931     AsyncUtil.runAsynchronously(new Runnable() {
932         public void run() { postServerCommand(command, arguments); });
933     }
934
935 private void postServerCommand(final String command, final YaillList
    arguments){
936     AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<
    JSONObject>() {
937         public void onSuccess(final JSONObject result) {
938             try {
939                 ServerCommandSuccess(command, JsonUtil.getListFromJsonArray(
    result.
940                     getJSONArray(MESSAGE_CONTENT_KEY)));
941             } catch (JSONException e) {
942                 Log.w(LOG_TAG, e);
943                 Info("Server command response failed to parse.");
944             }
945             FunctionCompleted("ServerCommand");
946         }
947
948         public void onFailure(String message) {
949             ServerCommandFailure(command, arguments);
950             WebServiceError("ServerCommand", message);
951         }
952     };
953
954     Log.d(LOG_TAG, "Going to post " + command + " with args " +
    arguments);
955     postCommandToGameServer(SERVER_COMMAND,
956         Lists.<NameValuePair>newArrayList(
957             new BasicNameValuePair(GAME_ID_KEY, GameId()),
958             new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
959             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
960             new BasicNameValuePair(COMMAND_TYPE_KEY, command),
961             new BasicNameValuePair(COMMAND_ARGUMENTS_KEY, arguments.
    toJSONString())),
962         myCallback);
963 }
964
965 /**
966  * Specifies the game instance id. Taken together, the game ID and
967  * the instance ID uniquely identify the game.
968  *
969  * @param instanceId the name of the game instance to join.
970  */
971 @SimpleFunction(description = "Sets InstanceId and joins the " +
972     "specified instance.")
973 public void SetInstance(final String instanceId) {
974     AsyncUtil.runAsynchronously(new Runnable() {
975         public void run() {
976             if (instanceId.equals("")) {
977                 Log.d(LOG_TAG, "Instance id set to empty string.");

```



```

978         if (!InstanceId().equals("")) {
979             instance = new GameInstance("");
980             InstanceIdChanged("");
981             FunctionCompleted("SetInstance");
982         }
983     } else {
984         postSetInstance(instanceId);
985     }
986 }
987 });
988 }
989
990 private void postSetInstance(String instanceId) {
991     AsyncCallbackPair<JSONObject> setInstanceCallback = new
992     AsyncCallbackPair<JSONObject>() {
993         public void onSuccess(final JSONObject response) {
994             processInstanceLists(response);
995             FunctionCompleted("SetInstance");
996         }
997         public void onFailure(final String message) {
998             WebServiceError("SetInstance", message);
999         }
1000     };
1001
1002     postCommandToGameServer(JOIN_INSTANCE_COMMAND,
1003         Lists.<NameValuePair>newArrayList(
1004             new BasicNameValuePair(GAME_ID_KEY, GameId()),
1005             new BasicNameValuePair(INSTANCE_ID_KEY, instanceId),
1006             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress())
1007         ),
1008         setInstanceCallback, true);
1009 }
1010
1011 /**
1012  * Specifies the game's leader. At any time, each game instance
1013  * has only one leader, but the leader may change over time.
1014  * Initially, the leader is the game instance creator. Application
1015  * inventors determine special properties of the leader.
1016  *
1017  * The leader can only be set by the current leader of the game.
1018  *
1019  * @param playerEmail a string containing the email address of the
1020  * player to become leader. The email should be in one of the
1021  * following formats:
1022  * <br>"Name O. Person &lt;name.o.person@gmail.com>"
1023  * <br>"name.o.person@gmail.com".
1024  */
1025 @SimpleFunction(description = "Tells the server to set the " +
1026     "leader to playerId. Only the current leader may " +
1027     "successfully set a new leader.")
1028 public void SetLeader(final String playerEmail) {
1029     AsyncUtil.runAsynchronously(new Runnable() {
1030         public void run() { postSetLeader(playerEmail); });
1031     }
1032 }

```

```

1030
1031 private void postSetLeader(final String newLeader) {
1032     AsyncCallbackPair<JSONObject> setLeaderCallback = new
1033     AsyncCallbackPair<JSONObject>() {
1034         public void onSuccess(final JSONObject response) {
1035             FunctionCompleted("SetLeader");
1036         }
1037         public void onFailure(final String message) {
1038             WebServiceError("SetLeader", message);
1039         }
1040     };
1041
1042     if (InstanceId().equals("")) {
1043         Info("You must join an instance before attempting to set a
1044         leader.");
1045         return;
1046     }
1047
1048     postCommandToGameServer(SET_LEADER_COMMAND,
1049         Lists.<NameValuePair>newArrayList(
1050             new BasicNameValuePair(GAME_ID_KEY, GameId()),
1051             new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
1052             new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
1053             new BasicNameValuePair(LEADER_KEY, newLeader)),
1054             setLeaderCallback);
1055 }
1056
1057 //-----
1058 // Activity Lifecycle Management
1059
1060 /**
1061  * Called automatically by the operating system.
1062  *
1063  * Currently does nothing.
1064  */
1065 public void onResume() {
1066     Log.d(LOG_TAG, "Activity Resumed.");
1067 }
1068
1069 /**
1070  * Called automatically by the operating system.
1071  *
1072  * Currently does nothing.
1073  */
1074 public void onStop() {
1075     Log.d(LOG_TAG, "Activity Stopped.");
1076 }
1077
1078 //-----
1079 // Utility Methods
1080
1081 private void postCommandToGameServer(final String commandName,
1082     List<NameValuePair> params, final AsyncCallbackPair<JSONObject>
1083     callback) {

```

```

1081     postCommandToGameServer(commandName, params, callback, false);
1082 }
1083
1084 private void postCommandToGameServer(final String commandName,
1085     final List<NameValuePair> params, final AsyncCallbackPair<
1086     JSONObject> callback,
1087     final boolean allowInstanceIdChange) {
1088     AsyncCallbackPair<JSONObject> thisCallback = new AsyncCallbackPair<
1089     <JSONObject>() {
1090         public void onSuccess(JSONObject responseObject) {
1091             Log.d(LOG_TAG, "Received response for " + commandName + ": " +
1092             responseObject.toString());
1093
1094             try {
1095                 if (responseObject.getBoolean(ERROR_RESPONSE_KEY)) {
1096                     callback.onFailure(responseObject.getString(
1097                     SERVER_RETURN_VALUE_KEY));
1098                 } else {
1099                     String responseGameId = responseObject.getString(
1100                     GAME_ID_KEY);
1101                     if (!responseGameId.equals(GameId())) {
1102                         Info("Incorrect game id in response: " + " +
1103                         responseGameId + ".");
1104                         return;
1105                     }
1106                     String responseInstanceId = responseObject.getString(
1107                     INSTANCE_ID_KEY);
1108                     if (responseInstanceId.equals("")) {
1109                         callback.onSuccess(responseObject.getJSONObject(
1110                         SERVER_RETURN_VALUE_KEY));
1111                         return;
1112                     }
1113                     if (responseInstanceId.equals(InstanceId())) {
1114                         updateInstanceInfo(responseObject);
1115                     } else {
1116                         if (allowInstanceIdChange || InstanceId().equals("")) {
1117                             instance = new GameInstance(responseInstanceId);
1118                             updateInstanceInfo(responseObject);
1119                             InstanceIdChanged(responseInstanceId);
1120                         } else {
1121                             Info("Ignored server response to " + commandName + "
1122                             for incorrect instance " +
1123                             responseInstanceId + ".");
1124                             return;
1125                         }
1126                     }
1127                     callback.onSuccess(responseObject.getJSONObject(
1128                     SERVER_RETURN_VALUE_KEY));
1129                 }
1130             } catch (JSONException e) {
1131                 Log.w(LOG_TAG, e);
1132                 callback.onFailure("Failed to parse JSON response to command
1133                 " + commandName);

```

```

1124     }
1125 }
1126 public void onFailure(String failureMessage) {
1127     Log.d(LOG_TAG, "Posting to server failed for " + commandName +
1128         " with arguments " +
1129         params + "\n Failure message: " + failureMessage);
1130     callback.onFailure(failureMessage);
1131 }
1132 };
1133 WebServiceUtil.getInstance().postCommandReturningObject (ServiceUrl
1134     (), commandName, params,
1135     thisCallback);
1136 }
1137 private void updateInstanceInfo(JSONObject responseObject) throws
1138     JSONException {
1139     boolean newLeader = false;
1140     String leader = responseObject.getString(LEADER_KEY);
1141     List<String> receivedPlayers = JsonUtil.getStringListFromJsonArray
1142     (responseObject.
1143         getJSONArray (PLAYERS_LIST_KEY));
1144     if (!Leader().equals(leader)) {
1145         instance.setLeader(leader);
1146         newLeader = true;
1147     }
1148     PlayerListDelta playersDelta = instance.setPlayers(receivedPlayers
1149 );
1150     if (playersDelta != PlayerListDelta.NO_CHANGE) {
1151         for (final String player : playersDelta.getPlayersRemoved()) {
1152             PlayerLeft(player);
1153         }
1154         for (final String player : playersDelta.getPlayersAdded()) {
1155             PlayerJoined(player);
1156         }
1157     }
1158     if (newLeader) {
1159         NewLeader(Leader());
1160     }
1161 }
1162 }

```

B.2: GameInstance.java - A container for information pertaining to game instances.

```

1 // Copyright 2009 Google Inc. All Rights Reserved.
2
3 package com.google.devtools.simple.runtime.components.android.util;
4
5 import java.util.ArrayList;
6 import java.util.HashMap;

```

```

7  import java.util.List;
8  import java.util.Map;
9
10 /**
11  * A container for information about a GameInstance for use
12  * with the App Inventor game framework.
13  *
14  * @author billmag@google.com (Bill Magnuson)
15  *
16  */
17 public class GameInstance {
18     private String instanceId;
19     private String leader;
20
21     // players in the current game
22     private List<String> players;
23
24     // Use this to store the most recent time stamp of each message type
25     // received.
26     private Map<String, String> messageTimes;
27
28     /**
29     * A GameInstance contains the most recent values
30     * for the leader and players of a particular game instance.
31     *
32     * This object is also used to keep track of the most recent
33     * time that a particular message type was retrieved from the
34     * server.
35     *
36     * @param instanceId The unique String that identifies this
37     * instance.
38     */
39     public GameInstance(String instanceId) {
40         players = new ArrayList<String>(0);
41         messageTimes = new HashMap<String, String>();
42         this.instanceId = instanceId;
43         this.leader = "";
44     }
45
46     /**
47     * Return the instance id of this instance.
48     * @return the instance id.
49     */
50     public String getInstanceId() {
51         return instanceId;
52     }
53
54     /**
55     * Return the current leader of this instance.
56     * @return The email address of the current leader.
57     */
58     public String getLeader() {
59         return leader;

```

```

60     }
61
62     /**
63      * Sets the leader of this instance.
64      * @param leader The email address of the new leader.
65      */
66     public void setLeader(String leader) {
67         this.leader = leader;
68     }
69
70     /**
71      * Sets the players of this instances to currentPlayersList.
72      *
73      * Compares the current players list with the new one and returns
74      * a delta to the caller.
75      *
76      * @param newPlayersList All players currently in the instance.
77      * @return PlayersListDelta.NO_CHANGE if there is no change in
78      * membership. Otherwise returns a PlayersListDelta with the
79      * appropriate player lists.
80      */
81     public PlayerListDelta setPlayers(List<String> newPlayersList) {
82         if (newPlayersList.equals(players)) {
83             return PlayerListDelta.NO_CHANGE;
84         }
85         List<String> removed = players;
86         List<String> added = new ArrayList<String>(newPlayersList);
87         players = new ArrayList<String>(newPlayersList);
88
89         added.removeAll(removed);
90         removed.removeAll(newPlayersList);
91         // This happens if the players list is the same but the ordering
92         // has changed for some reason.
93         if (added.size() == 0 && removed.size() == 0) {
94             return PlayerListDelta.NO_CHANGE;
95         }
96
97         return new PlayerListDelta(removed, added);
98     }
99
100    /**
101     * Return the list of players currently in this instance.
102     *
103     * @return A list of the players in the instance.
104     */
105    public List<String> getPlayers() {
106        return players;
107    }
108
109    /**
110     * Return the most recently put time string for this type.
111     *
112     * This should represent the creation time of the most
113     * recently received message of the specified type and can

```

```

114     * be used to filter available messages to find those that
115     * have not been received.
116     *
117     * @param type The message type.
118     * @return The most recently put value for this type.
119     */
120     public String getMessageTime(String type) {
121         if (messageTimes.containsKey(type)) {
122             return messageTimes.get(type);
123         }
124         return "";
125     }
126
127     /**
128     * Puts a new time string for the specified message type.
129     *
130     * The string should be some value that can be understood
131     * by its eventual consumer. It is left as a string here
132     * to remove the need to convert back and forth from DateTime
133     * objects when dealing with web services.
134     *
135     * @param type The message type.
136     * @param time A string representing the time the message
137     * was created.
138     */
139     public void putMessageTime(String type, String time) {
140         messageTimes.put(type, time);
141     }
142 }

```

B.2 Utilities and Data Structures

B.3: YailList.java - The AppInventor collection primitive.

```

1  // Copyright 2009 Google Inc. All Rights Reserved.
2
3  package com.google.devtools.simple.runtime.components.util;
4
5  import com.google.devtools.simple.runtime.errors.YailRuntimeError;
6
7  import java.util.List;
8  import java.util.Collection;
9
10 import gnu.lists.FString;
11 import gnu.lists.LList;
12 import gnu.lists.Pair;
13
14 import org.json.JSONException;
15 import org.json.JSONObject;
16
17 /**

```

```

18  * The YailList is a wrapper around the gnu.list.Pair class used
19  * by the Kawa framework. YailList is the main list primitive used
20  * by App Inventor components.
21  *
22  * @author gleitz@google.com (Benjamin Gleitzman)
23  * @author billmag@google.com (Bill Magnuson)
24  */
25  public class YailList extends Pair {
26
27      // Component writers take note!
28      // If you want to pass back a list to the blocks language, the
29      // straightforward way to do this is simply to pass
30      // back an ArrayList. If you construct a YailList to return
31      // to codeblocks, you must guarantee that the elements of the list
32      // are "sanitized". That is, you must pass back a tree whose
33      // subtrees are themselves YailLists, and whose leaves are all
34      // legitimate Yail data types. See the definition of sanitization
35      // in runtime.scm.
36
37      /**
38       * Create an empty YailList.
39       */
40      public YailList() {
41          super(YailConstants.YAIL_HEADER, LList.Empty);
42      }
43
44      private YailList(Object cdrval) {
45          super(YailConstants.YAIL_HEADER, cdrval);
46      }
47
48      /**
49       * Create a YailList from an array.
50       */
51      public static YailList makeList(Object[] objects) {
52          LList newCdr = Pair.makeList(objects, 0);
53          return new YailList(newCdr);
54      }
55
56      /**
57       * Create a YailList from a List.
58       */
59      public static YailList makeList(List vals) {
60          LList newCdr = Pair.makeList(vals);
61          return new YailList(newCdr);
62      }
63
64      /**
65       * Create a YailList from a Collection.
66       */
67      public static YailList makeList(Collection vals) {
68          LList newCdr = Pair.makeList(vals.toArray(), 0);
69          return new YailList(newCdr);
70      }
71

```



```

72  /**
73   * Return this YailList as an array.
74   */
75  @Override
76  public Object[] toArray() {
77      if (cdr instanceof Pair) {
78          return ((Pair) cdr).toArray();
79      } else if (cdr instanceof LList) {
80          return ((LList) cdr).toArray();
81      } else {
82          throw new YailRuntimeError("YailList cannot be represented as an
83              array", "YailList Error.");
84      }
85  }
86  /**
87   * Return this YailList as an array of Strings.
88   */
89  public String[] toStringArray() {
90      int size = this.size();
91      String[] objects = new String[size];
92      for (int i = 1; i <= size; i++) {
93          objects[i - 1] = String.valueOf(get(i));
94      }
95      return objects;
96  }
97
98  /**
99   * Return a strictly syntactically correct JSON text
100   * representation of this YailList. Only supports String, Number,
101   * Boolean, YailList, FString and arrays containing these types.
102   */
103  public String toJSONString() {
104      try {
105          StringBuilder json = new StringBuilder();
106          String separator = "";
107          json.append('[');
108          int size = this.size();
109          for (int i = 1; i <= size; i++) {
110              Object value = get(i);
111              json.append(separator).append(getJsonRepresentation(value));
112              separator = ", ";
113          }
114          json.append(']');
115
116          return json.toString();
117      } catch (JSONException e) {
118          throw new YailRuntimeError("List failed to convert to JSON.", "
119              JSON Creation Error.");
120      }
121  }
122
123  /**

```

```

124     * Return the size of this YailList.
125     */
126     @Override
127     public int size() {
128         return super.size() - 1;
129     }
130
131     /**
132     * Return a String representation of this YailList.
133     */
134     @Override
135     public String toString() {
136         if (cdr instanceof Pair) {
137             return ((Pair) cdr).toString();
138         } else if (cdr instanceof LList) {
139             return ((LList) cdr).toString();
140         } else {
141             throw new RuntimeException("YailList cannot be represented as a
String");
142         }
143     }
144
145     /**
146     * Return the String at the given index.
147     */
148     public String getString(int index) {
149         return (String) get(index + 1);
150     }
151
152     private String getJsonRepresentation(Object value) throws
JSONException {
153         if (value == null || value.equals(null)) {
154             return "null";
155         }
156         if (value instanceof FString) {
157             return JSONObject.quote(value.toString());
158         }
159         if (value instanceof YailList) {
160             return ((YailList) value).toJSONString();
161         }
162         if (value instanceof Number) {
163             return JSONObject.numberToString((Number) value);
164         }
165         if (value instanceof Boolean) {
166             return value.toString();
167         }
168         if (value.getClass().isArray()) {
169             StringBuilder sb = new StringBuilder();
170             sb.append("[");
171             String separator = "";
172             for (Object o: (Object[]) value) {
173                 sb.append(separator).append(getJsonRepresentation(o));
174                 separator = ", ";
175             }

```

```

176         sb.append("]");
177         return sb.toString();
178     }
179     return JSONObject.quote(value.toString());
180 }
181
182
183 }

```

B.4: JsonUtil.java - Utility functions for converting JSON to data representations understood by AppInventor.

```

1  // Copyright 2010 Google Inc. All Rights Reserved.
2
3  package com.google.devtools.simple.runtime.components.util;
4
5  import org.json.JSONArray;
6  import org.json.JSONException;
7  import org.json.JSONObject;
8
9  import java.util.ArrayList;
10 import java.util.Collections;
11 import java.util.Iterator;
12 import java.util.List;
13
14 /**
15  * Provides utility functions to create Java collections out of
16  * JSON.
17  *
18  * @author billmag@google.com (Bill Magnuson)
19  */
20
21 public class JsonUtil {
22
23     /**
24      * Prevent instantiation.
25      */
26     private JsonUtil() {
27     }
28
29     /**
30      * Returns a list of String objects from a JSONArray. This
31      * does not do any kind of recursive unpacking of the array.
32      * Thus, if the array includes other JSON arrays or JSON objects
33      * their string representation will be a single item in the
34      * returned list.
35      *
36      * @param jArray The JSONArray to convert.
37      * @return A List of the String representation of each item in
38      * the JSON array.
39      * @throws JSONException if an element of jArray cannot be
40      * converted to a String.

```

```

41     */
42     public static List<String> getStringListFromJsonArray(JSONArray
    jsonArray) throws JSONException {
43         List<String> returnList = new ArrayList<String>();
44         for (int i = 0; i < jsonArray.length(); i++) {
45             String val = jsonArray.getString(i);
46             returnList.add(val);
47         }
48         return returnList;
49     }
50
51     /**
52      * Returns a Java Object list of a JSONArray with each item in
53      * the array converted using convertJsonItem().
54      *
55      * @param jsonArray The JSONArray to convert.
56      * @return A List of Strings and more Object lists.
57      * @throws JSONException if an element in jsonArray cannot be
58      * converted properly.
59      */
60     public static List<Object> getListFromJsonArray(JSONArray jsonArray)
    throws JSONException {
61         List<Object> returnList = new ArrayList<Object>();
62         for (int i = 0; i < jsonArray.length(); i++) {
63             returnList.add(convertJsonItem(jsonArray.get(i)));
64         }
65         return returnList;
66     }
67
68     /**
69      * Returns a list containing one two item list per key in jobject.
70      * Each two item list has the key String as its first element and
71      * the result of calling convertJsonItem() on its value as the
72      * second element. The sub-lists in the returned list will appear
73      * in alphabetical order by key.
74      *
75      * @param jobject The JSONObject to convert.
76      * @return A list of two item lists: [String key, Object value].
77      * @throws JSONException if an element in jobject cannot be
78      * converted properly.
79      */
80     public static List<Object> getListFromJsonObject(JSONObject jobject)
    throws JSONException {
81         List<Object> returnList = new ArrayList<Object>();
82         Iterator<String> keys = jobject.keys();
83
84         List<String> keysList = new ArrayList<String>();
85         while (keys.hasNext()) {
86             keysList.add(keys.next());
87         }
88         Collections.sort(keysList);
89
90         for (String key : keysList) {
91             List<Object> nestedList = new ArrayList<Object>();

```

```

92     nestedList.add(key);
93     nestedList.add(convertJsonItem(jObject.get(key)));
94     returnList.add(nestedList);
95 }
96
97 return returnList;
98 }
99
100 /**
101  * Returns a Java object representation of objects that are
102  * encountered inside of JSON created using the org.json package.
103  * JSON arrays and objects are transformed into their list
104  * representations using getListFromJsonArray and
105  * getListFromJsonObject respectively.
106  *
107  * Java Boolean values and the Strings "true" and "false" (case
108  * insensitive) are inserted as Booleans. Java Numbers are
109  * inserted without modification and all other values are inserted
110  * as their toString(). value.
111  *
112  * @param o An item in a JSON array or JSON object to convert.
113  * @return A Java Object representing o or the String "null"
114  * if o is null.
115  * @throws JSONException if o fails to parse.
116  */
117 public static Object convertJsonItem(Object o) throws JSONException
118 {
119     if (o == null) {
120         return "null";
121     }
122     if (o instanceof JSONObject) {
123         return getListFromJsonObject((JSONObject) o);
124     }
125     if (o instanceof JSONArray) {
126         return getListFromJsonArray((JSONArray) o);
127     }
128     if (o.equals(Boolean.FALSE) || (o instanceof String &&
129         ((String) o).equalsIgnoreCase("false"))) {
130         return false;
131     }
132     if (o.equals(Boolean.TRUE) || (o instanceof String && ((String) o)
133         .equalsIgnoreCase("true"))) {
134         return true;
135     }
136     if (o instanceof Number) {
137         return o;
138     }
139     return o.toString();
140 }

```

```
144     }
145 }
```

B.5: WebServiceUtil.java - Utility functions for making POST commands from Android applications.

```
1  // Copyright 2009 Google Inc. All Rights Reserved.
2
3  package com.google.devtools.simple.runtime.components.android.util;
4
5  import android.util.Log;
6
7  import org.apache.http.NameValuePair;
8  import org.apache.http.client.ClientProtocolException;
9  import org.apache.http.client.HttpClient;
10 import org.apache.http.client.ResponseHandler;
11 import org.apache.http.client.entity.UrlEncodedFormEntity;
12 import org.apache.http.client.methods.HttpPost;
13 import org.apache.http.conn.params.ConnManagerParams;
14 import org.apache.http.conn.scheme.PlainSocketFactory;
15 import org.apache.http.conn.scheme.Scheme;
16 import org.apache.http.conn.scheme.SchemeRegistry;
17 import org.apache.http.conn.ssl.SSLSocketFactory;
18 import org.apache.http.impl.client.BasicResponseHandler;
19 import org.apache.http.impl.client.DefaultHttpClient;
20 import org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager;
21 import org.apache.http.params.BasicHttpParams;
22 import org.apache.http.params.HttpConnectionParams;
23 import org.apache.http.protocol.HTTP;
24 import org.json.JSONArray;
25 import org.json.JSONException;
26 import org.json.JSONObject;
27
28 import java.io.IOException;
29 import java.io.UnsupportedEncodingException;
30 import java.util.ArrayList;
31 import java.util.List;
32
33 /**
34  * These commands post to the Web and get responses that are assumed
35  * to be JSON structures: a string, a JSON array, or a JSON object.
36  * It's up to the caller of these routines to decide which version
37  * to use, and to decode the response.
38  *
39  * @author halabelson@google.com (Hal Abelson)
40  * @author billmag@google.com (Bill Magnuson)
41  */
42 public class WebServiceUtil {
43
44     private static final WebServiceUtil INSTANCE = new WebServiceUtil();
45     private static final String LOG_TAG = "WebServiceUtil";
46     private static HttpClient httpClient = null;
```

```

47 private static Object httpClientSynchronizer = new Object();
48
49 private WebServiceUtil(){
50 }
51
52 /**
53  * Returns the one <code>WebServiceUtil</code> instance
54  * @return the one <code>WebServiceUtil</code> instance
55  */
56 public static WebServiceUtil getInstance() {
57     // This needs to be here instead of in the constructor because
58     // it uses classes that are in the AndroidSDK and thus would
59     // cause Stub! errors when running the component descriptor.
60     synchronized(httpClientSynchronizer) {
61         if (httpClient == null) {
62             SchemeRegistry schemeRegistry = new SchemeRegistry();
63             schemeRegistry.register(new Scheme("http", PlainSocketFactory.
64 getSocketFactory(), 80));
65             schemeRegistry.register(new Scheme("https", SSLSocketFactory.
66 getSocketFactory(), 443));
67             BasicHttpParams params = new BasicHttpParams();
68             HttpConnectionParams.setConnectionTimeout(params, 20 * 1000);
69             HttpConnectionParams.setSoTimeout(params, 20 * 1000);
70             ConnManagerParams.setMaxTotalConnections(params, 20);
71             ThreadSafeClientConnManager manager = new
72 ThreadSafeClientConnManager(params,
73             schemeRegistry);
74             WebServiceUtil.httpClient = new DefaultHttpClient(manager,
75 params);
76         }
77     }
78     return INSTANCE;
79 }
80
81 /**
82  * Make a post command to serviceURL with params and return the
83  * response String as a JSON array.
84  *
85  * @param serviceURL The URL of the server to post to.
86  * @param commandName The path to the command.
87  * @param params A List of NameValuePairs to send as parameters
88  * with the post.
89  * @param callback A callback function that accepts a JSON array
90  * on success.
91  */
92 public void postCommandReturningArray(String serviceURL, String
93 commandName,
94     List<NameValuePair> params, final AsyncCallbackPair<JSONArray>
95 callback) {
96     AsyncCallbackPair<String> thisCallback = new AsyncCallbackPair<
97 String>() {
98         public void onSuccess(String httpResponseString) {
99             try {
100                 callback.onSuccess(new JSONArray(httpResponseString));
101             }
102         }
103     };
104 }

```

```

94         } catch (JSONException e) {
95             callback.onFailure(e.getMessage());
96         }
97     }
98     public void onFailure(String failureMessage) {
99         callback.onFailure(failureMessage);
100     }
101 };
102 postCommand(serviceURL, commandName, params, thisCallback);
103 }
104
105 /**
106  * Make a post command to serviceURL with paramaterss and
107  * return the response String as a JSON object.
108  *
109  * @param serviceURL The URL of the server to post to.
110  * @param commandName The path to the command.
111  * @param params A List of NameValuePairs to send as parameters
112  * with the post.
113  * @param callback A callback function that accepts a JSON object
114  * on success.
115  */
116 public void postCommandReturningObject(final String serviceURL, final
    String commandName,
117     List<NameValuePair> params, final AsyncCallbackPair<JSONObject>
callback) {
118     AsyncCallbackPair<String> thisCallback = new AsyncCallbackPair<
String>() {
119         public void onSuccess(String httpResponseString) {
120             try {
121                 callback.onSuccess(new JSONObject(httpResponseString));
122             } catch (JSONException e) {
123                 callback.onFailure(e.getMessage());
124             }
125         }
126         public void onFailure(String failureMessage) {
127             callback.onFailure(failureMessage);
128         }
129     };
130     postCommand(serviceURL, commandName, params, thisCallback);
131 }
132
133 /**
134  * Make a post command to serviceURL with params and return the
135  * response String.
136  *
137  * @param serviceURL The URL of the server to post to.
138  * @param commandName The path to the command.
139  * @param params A List of NameValuePairs to send as parameters
140  * with the post.
141  * @param callback A callback function that accepts a String on
142  * success.
143  */

```



```

144 public void postCommand(final String serviceURL, final String
    commandName,
145     List<NameValuePair> params, AsyncCallbackPair<String> callback)
    {
146     Log.d(LOG_TAG, "Posting " + commandName + " to " + serviceURL + "
        with arguments " + params);
147
148     if (serviceURL == null || serviceURL.equals("")) {
149         callback.onFailure("No service url to post command to.");
150     }
151     final HttpPost httpPost = new HttpPost(serviceURL + "/" +
        commandName);
152
153     if (params == null) {
154         params = new ArrayList<NameValuePair>();
155     }
156     try {
157         String httpResponseString;
158         ResponseHandler<String> responseHandler = new
        BasicResponseHandler();
159         httpPost.setEntity(new UrlEncodedFormEntity(params, HTTP.UTF_8))
        ;
160         httpPost.setHeader("Accept", "application/json");
161         httpResponseString = httpClient.execute(httpPost,
        responseHandler);
162         callback.onSuccess(httpResponseString);
163     } catch (UnsupportedEncodingException e) {
164         Log.w(LOG_TAG, e);
165         callback.onFailure("Failed to encode params for web service call
        .");
166     } catch (ClientProtocolException e) {
167         Log.w(LOG_TAG, e);
168         callback.onFailure("Communication with the web service
        encountered a protocol exception.");
169     } catch (IOException e) {
170         Log.w(LOG_TAG, e);
171         callback.onFailure("Communication with the web service timed out
        .");
172     }
173 }
174 }

```

Bibliography

- [1] Hal Abelson. App inventor for android.
<http://googleresearch.blogspot.com/2009/07/app-inventor-for-android.html>, July 2009.
- [2] Wen-Chih Chang, Te-Hua Wang, Freya H. Lin, and Hsuan-Che Yang. Game-based learning with ubiquitous technologies. *IEEE Internet Computing*, 13:26–33, 2009.
- [3] Microsoft Corporation. Kodu - microsoft research.
<http://research.microsoft.com/en-us/projects/kodu/>, January 2010.
- [4] Wanda Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice*. Prentice Hall, 2009.
- [5] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. Can mobile gaming be improved? In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 44, New York, NY, USA, 2006. ACM.
- [6] No Author Given. Installation and quick start - nose v0.11.1 documentation. <http://somethingaboutorange.com/mrl/projects/nose/0.11.1/>, February 2010.
- [7] No Author Given. Nosegae: Test support for google application engine. <http://farmdev.com/projects/nosegae/>, February 2010.
- [8] Björn Knutsson, Massively Multiplayer Games, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games, 2004.
- [9] Stan Kurkovsky. Engaging students through mobile game development. *SIGCSE Bull.*, 41(1):44–48, 2009.
- [10] C. McCaffrey. Starlogo tng: The convergence of graphical programming and text processing. Master's thesis, Massachusetts Institute of Technology, 2006.
- [11] R. Roque. Openblocks: An extendable framework for graphical block programming systems. Master's thesis, Massachusetts Institute of Technology, 2007.

- [12] Jason O. B. Soh and Bernard C. Y. Tan. Mobile gaming. *Commun. ACM*, 51(3):35–39, 2008.
- [13] Holly Stevens. Gartner says grey-market sales and destocking drive world-wide mobile phone sales to 309 million units; smartphone sales grew 13 per cent in third quarter of 2009, November 2009.