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 Building Blocks for Mobile Games multiplayer 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 Game Server provides basic functionality that can be used to create simple multiplayer games and message-passing applications, such as a multiuser bulletin board. The Game Server is also extensible and can be enhanced with custom modules which provide server commands that implement game logic, perform database operations, access third-party web services, and read RSS feeds. Custom modules were used with Building Blocks to develop a multiplayer card game, a variant 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. The clients for these applications are built entirely with the App Inventor graphical blocks language, which can be assembled into Android Applications. The custom modules that support the client programs average less than 50 lines of Python code.

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 offering his incredibly valuable advice throughout the process.

I also owe a debt of 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 are 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
   1.1 App Inventor for Android ............................. 20
   1.2 Games in App Inventor ............................... 21
   1.3 Thesis Summary .................................. 23

2 Introduction to App Inventor ............................. 25
   2.1 The Application Designer ............................. 25
      2.1.1 Non-Visible Components ......................... 27
   2.2 The Blocks Editor ................................ 29
      2.2.1 Property Getters ................................. 31
      2.2.2 Property Setters ................................. 31
      2.2.3 Event Handlers ................................ 31
      2.2.4 Method Calls ................................ 32
   2.3 Overview of Building Blocks for Mobile Games ........ 32
   2.4 Packaging and Running the Application .............. 35

3 Building a Game ........................................ 37
   3.1 User Interfaces .................................... 38
   3.2 The Voting Server Module ............................ 40
      3.2.1 Server Commands in the Game Client Component .. 42
      3.2.2 Defining Server Commands on the Game Server .. 43
      3.2.3 The Message Model ............................... 44
      3.2.4 Input Validation and Error Handling ............ 46
6.2 Mobile Games in Education ........................................... 95
6.3 Alternatives to Client-Server Design .............................. 97

7 Extensions ........................................................................ 99
7.1 User Interfaces and Multiple Screens ............................... 99
7.2 Saving Local State .......................................................... 100
7.3 Pushing Messages and Game State Updates ....................... 100

8 Contributions .................................................................. 103

A Game Server Code .......................................................... 105
A.1 Game Server ................................................................. 105
A.1.1 Models ................................................................. 137
A.1.2 Extensions .............................................................. 146
A.2 Custom Modules .......................................................... 161
A.2.1 Amazon ................................................................. 162
A.2.2 Androids to Androids ............................................... 164
A.2.3 Bulls and Cows ....................................................... 171
A.2.4 Voting ................................................................. 174

B Game Client Code .......................................................... 181
B.1 Game Client Component ............................................... 181
B.2 Utilities and Data Structures ......................................... 207
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). .................................................. 26

2-2 The completed user interface of a simple application for joining shared bulletin boards and posting messages for other users to see. ............. 27

2-3 The configurable properties for a button component shown in the application designer. ................................................................. 28

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. ......... 29

2-5 The built-in block drawer for list handling operations. Blocks that 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. ............................................... 30

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. ............................................ 31

2-7 An event handler for a button that pops up an input dialog when clicked. ................................................................. 32
The architecture of App Inventor for Android with the Building Blocks for Mobile Games multiplayer framework.  

The method call and event handler blocks for the Game Client component.  

The property getter blocks for the Game Client component. The properties provide access to information about the current configuration of the Game Client component, and the state of the current game instance.  

The process of creating and voting on polls with Mobile Voting.  

User interfaces for poll creation and voting.  

Viewing vote totals on both the poll creation and voting screens.  

When a user hits the “Submit Vote” button in Ballot Box, a request is made to invoke a Cast Vote server command. The arguments to the command are the poll ID number, which is accessed from the currentPoll global variable, and the index of the selected option.  

The method call block for sending a message to other players using the Game Client component.  

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.  

Event handlers for retrieving the list of polls owned by the user. After selecting a category, the Game Client 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.
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.

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.

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.

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.

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.

Viewing the FreeFood bulletin board.

The Bulletin Board message reading loop. Bulletin Board uses a Clock component to make message requests every 10 seconds.

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.
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.

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 the server.

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.

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.

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.

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.

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 him or her.
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.

5-12 The GotMessage event handler in Androids to Androids. Depending on the type of 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.

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.

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.

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]
## Code Listings

3.1 The voting module’s server command to create a new poll. (Excerpt from A.13) .................................................. 44

3.2 The command dictionary for four different custom modules. (Excerpt from A.9) ................................................. 46

4.1 Adding dynamic properties to a Message object. (Excerpt from A.13) ................................................. 59

5.1 The MoBulls and Cows server command to submit a new guess. Input validation and game ending code has been omitted for brevity. At the end of the method, the last guess and reply are saved as dynamic properties of the Message object that stores the MoBulls and Cows game information. If a subsequent guess has the exact same arguments, the computation and score deduction are skipped. Instead, the saved reply is immediately returned. (Excerpt from A.12) ................................................. 78

5.2 Server code required to perform a search by keyword. (Excerpt from A.10) .................................................. 83

5.3 The custom server command for submitting a noun card to the leader. (Excerpt from A.11) .................................................. 86

A.1 server.py - The Game Server application file. Includes the request handlers for server requests. .................................................. 105

A.2 server.commands.py - Contains the server command dictionary and built-in server commands relating to instance management. .................................................. 129

A.3 utils.py - Helper utility functions for input sanitizing and database operations. .................................................. 134

A.4 game.py - The game database model. .................................................. 137
<table>
<thead>
<tr>
<th>Section</th>
<th>File Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.5</td>
<td>game_instance.py</td>
<td>The game instance database model.</td>
<td>139</td>
</tr>
<tr>
<td>A.6</td>
<td>message.py</td>
<td>The message database model.</td>
<td>144</td>
</tr>
<tr>
<td>A.7</td>
<td>card_game.py</td>
<td>A library for handling card games in a game instance.</td>
<td>146</td>
</tr>
<tr>
<td>A.8</td>
<td>scoreboard.py</td>
<td>A library for keeping track of a per instance score-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>board.</td>
<td>156</td>
</tr>
<tr>
<td>A.9</td>
<td>commands.py</td>
<td>The command dictionary for custom modules.</td>
<td>161</td>
</tr>
<tr>
<td>A.10</td>
<td>amazon_commands.py</td>
<td>Amazon server commands.</td>
<td>162</td>
</tr>
<tr>
<td>A.11</td>
<td>ata_commands.py</td>
<td>Androids to Androids game commands.</td>
<td>164</td>
</tr>
<tr>
<td>A.12</td>
<td>bac_commands.py</td>
<td>Bulls and Cows game commands.</td>
<td>171</td>
</tr>
<tr>
<td>A.13</td>
<td>voting_commands.py</td>
<td>Voting commands.</td>
<td>174</td>
</tr>
<tr>
<td>B.1</td>
<td>GameClient.java</td>
<td>The Game Client component.</td>
<td>181</td>
</tr>
<tr>
<td>B.2</td>
<td>GameInstance.java</td>
<td>A container for information pertaining to game</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>instances.</td>
<td>204</td>
</tr>
<tr>
<td>B.3</td>
<td>YailList.java</td>
<td>The App Inventor collection primitive.</td>
<td>207</td>
</tr>
<tr>
<td>B.4</td>
<td>JsonUtil.java</td>
<td>Utility functions for converting JSON to data repre-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sentations understood by App Inventor.</td>
<td>211</td>
</tr>
<tr>
<td>B.5</td>
<td>WebServiceUtil.java</td>
<td>Utility functions for making POST commands</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>from Android applications.</td>
<td>214</td>
</tr>
</tbody>
</table>
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, smart phone ownership is on the rise and even most teenagers now own mobile phones[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. People are unable to create programs that are customized for their own life or implement the great ideas they have for programs because the process of building them is extremely difficult and requires background knowledge of Computer Science topics.

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, multiuser 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 multiuser applications very difficult. To address these issues and empower mobile phone users to create social applications and games, I created the Building Blocks for Mobile Games multiplayer framework for App Inventor for Android. Building Blocks for Mobile Games 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 Building Blocks for Mobile Games 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 gives 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 components are exposed to application developers via graphical code blocks, instead of with written code. Just like putting together a puzzle, users of App Inventor can snap together 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. Building Blocks for Mobile Games widens this focus by creating a Game Client 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 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 Game Client component follows the pattern of existing App Inventor components by packaging code to perform procedures or to handle program events into graphical blocks which can be used by the program creator to create applications. When the Game Client’s blocks are used to call procedures, the component sends requests using the phone’s mobile data connection to the Game Server\(^1\). 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

---

\(^1\)The 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.
as when a player enters or leaves a game and when an invite to join another game has been received.

Without modification, the Game Server 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 very important because the client-side component can only be modified by changing the production App Inventor servers run by Google. Thus, to allow applications to access external resources and functionality which is not available in App Inventor, the Game Server was built to provide extensibility through custom modules that can be added to a Game Server installation and accessed with the Game Client component.

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. A class at Wellesley, even worked with the Google team to create a custom component that allowed applications to access a server that implemented a voting application\(^2\). 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 App Engine server. 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 walks through a reimplementation of the voting application built at Wellesley. The new version requires only that a single Python file with less than 100 lines of code be added to the Game Server. On the App Inventor server, no new components or other modifications are needed.

\(^2\)The course, CS 114 - Technologies for Communication was taught in the fall of 2009 by Professors Takis Metaxas, Eni Mustafaraj and Lyn Turbak. The custom component was created primarily by Prof. Mustafaraj.
1.3 Thesis Summary

Chapter 2 provides an introduction to the App Inventor system. Chapter 3 is a walkthrough of the creation of a voting application. An in-depth look at the Game Client component and the server API are given in the system overview in Chapter 4.

After the system overview, Chapter 5 describes four other example applications that I have built to demonstrate the various use patterns of the Game Client component and Game Server:

- Bulletin Board - A multiuser online message board which was created without any modifications to the Game Server.

- MoBulls and Cows - A version of the classic pen-and-paper game Bulls and Cows which implements game logic in a custom module and uses the Game Server to maintain a high score list among all players of the game.

- Amazon - A book lookup application which accesses Amazon's E-Commerce Services with a custom module.

- Androids to Androids - A multiplayer card game which shows players simultaneously participating in a multiple round card game with score keeping.

Then, Chapter 6 presents other work in the areas of graphical programming and mobile game creation. Chapters 7 and 8 discuss possible future extensions to Building Blocks for Mobile Applications and list the contributions of this thesis.
Chapter 2

Introduction to App Inventor

This chapter gives an introduction to the two-part process for building Android applications 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 Building Blocks for Mobile Games 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
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).

them from the palette on the left to the viewer in the middle. Figure 2-2 shows a 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.
2.1.1 Non-Visible Components

In addition to the user interface widgets, the designer also includes non-visible components. Each non-visible component 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 Game Client component, a web database with simple put and get operations, a Twitter component and the original Voting component.

The Game Client component includes blocks for completing requests to the Game Server. When a call is made using the Game Client, a new connection is opened to the Game Server and a POST request is made. The Game Server reply is interpreted by the Game Client and used to update its own properties or trigger events related to the game.
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 that 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.
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. 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 Overview of Building Blocks for Mobile Games

The Building Blocks for Mobile Games multiplayer framework consists of a non-visible Game Client 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 Game Client component and use the method call blocks to make server requests. The basic architecture of this system is shown in Figure 2-8.

The Game Client includes built-in method call blocks which make requests to perform actions such as joining game instances, inviting new members, sending
Application development is done in an online environment that runs from the App Inventor server. Using the GameClient component, multiplayer applications send requests to the game server running on App Engine. 

Applications store game state and messages in databases on App Engine.

Figure 2-8: The architecture of App Inventor for Android with the Building Blocks for Mobile Games multiplayer framework.

messages to other players and executing custom server commands. The method call blocks are shown in Figure 2-9 (a). The Game Server defines a request handler for each of these actions which accept POST requests from the Game Client and reply with JSON objects containing the result of executing the server command. 

When the server reply is received, the Game Client processes it and triggers events based on the result of the request, and in response to any changes to the game state on the server. Nine of the event handler blocks are shown in Figure 2-9 (b). NewLeader, PlayerJoined and PlayerInvited events occur when the game state changes on the server. GotMessage triggers for each message received after a GetMessages method call block is executed. Chapter 4 provides a detailed description of the method call blocks, server API and event handlers of Building Blocks for Mobile Games.

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 iden-
(a) The method call blocks for the Game Client component. Each block makes a request to the Game Server and processes the result to trigger appropriate events.

(b) A selection of Game Client event handlers. Events are triggered to indicate that game state has changed or to provide useful information from server responses.

Figure 2-9: The method call and event handler blocks for the Game Client component.
tified by the game ID property of its Game Client 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 the Game Server inside of custom server commands.

In addition to the method calls and event handlers, the Game Client provides read access to its properties including the list of players currently in a game instance, lists of instances the current player has joined and been invited to, and the email address of the current player. These property blocks are shown in Figure 2-10.

The Game Server can also be augmented with custom modules written by application creators that define server commands to be called with the Game Client’s ServerCommand block. These custom modules have full access to the App Engine database 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

Once the application is designed and its blocks have been defined, it needs to be packaged by the App Inventor server into an installable application. The user
Figure 2-10: The property getter blocks for the Game Client component. The properties provide access to information about the current configuration of the Game Client component, and the state of the current game instance.

This process kicks off 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 project. 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 multiuser voting application called Mobile Voting using Building Blocks for Mobile Games and App Inventor for Android. The chapter starts with a presentation of the user interface design, moves to a discussion of the custom server module and finally shows the blocks that make up the client. A voting application is uniquely suited to demonstrate a multiplayer framework because it uses interaction with others to derive enjoyment and utility. Mobile Voting consists of three parts:

- **Ballot Box program** - 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 Creator program** - 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 Game Client component by using a method call block.

With Mobile Voting, users can create polls using the Poll Creator on their Android phones. These polls are uploaded to the Game Server and stored in the App
New polls are sent to the Game Server.

Poll results are updated as new votes are submitted.

Polls and votes are stored in the App Engine datastore.

Ballot Box retrieves new polls from the Game Server. Users then decide on the polls and submit their votes.

Figure 3-1: The process of creating and voting on polls with Mobile Voting.

Engine database. Other users can then use the Ballot Box program to vote on polls that have been made by others. Both the Ballot Box program and the Poll Creator program are created entirely using App Inventor with the Game Client component. The processes of creating polls and casting votes are shown in Figure 3-1.

3.1 User Interfaces

In order to simplify the user interface, Mobile Voting splits poll management and voting into two separate Android programs\(^1\). The user interface for each program is shown in Figure 3-2. Together, the two programs provide a user with the ability to do the following:

\(^1\)App 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.
• 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 3-3 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.

Polls are split into different categories depending on their subject. All of the polls hosted on the demonstration server are made public, so that any player may join them. Players select from available lists and categories by using a component called a ListPicker. ListPickers 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 3-3 (b), where the “Submit Vote” button is disabled before a user
Please input the options for your poll. You must have at least two to submit.

Current Category: Miscellaneous

Question: What should I name my dog?

1: Bob
2: Sir barks a lot.
3: Cat
4: 

You have not voted in this poll yet.

Submit Vote

(b) The Mobile Voting Ballot Box. 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 3-2: User interfaces for poll creation and voting.

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
Question: What should I name my dog?

1: (75 votes) Bob
2: (42 votes) Sir barks a lot.
3: (31 votes) Cat
4: 
5: 

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

The 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 commands 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.

3.2.1 Server Commands in the Game Client Component

Figure 3-4: When a user hits the “Submit Vote” button in Ballot Box, a request is made to invoke a Cast Vote server command. The arguments to the command are the poll ID number, which is accessed from the currentPoll global variable, and the index of the selected option.
Requests to execute server commands are made by using the Game Client component’s ServerCommand block. The server command block accepts two parameters:

- **Command** - The key for the requested server command. The Game Server uses a map of server command keys to custom module functions\(^2\) to find the requested function and invoke it with the custom arguments defined in the second parameter.

- **Arguments** - A list of arguments to pass to the server command. The makeup of the list of arguments varies by command. This generic handling of arguments allows for server commands to accept any number of arguments of different types without modifying the Game Client component.

An example of a ServerCommand block being used to execute the Cast Vote server command is shown in Figure 3-4.

### 3.2.2 Defining Server Commands on the Game Server

On the Game Server, 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 GameInstance database model. The Game Server uses game 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

\(^2\)This map is explained in more detail in Section 3.2.5.
options has been provided. The procedure then initializes a Message object with an empty recipient and stores it in the database.³

---

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

```python
3 def make_new_poll_command(instance, player, arguments):
    """Make a new poll.

    Args:
    - instance: The game instance to add the poll to.
    - player: The email of the player creating the poll.
    - arguments: A two-item list containing the question and a second list of 2-5 options.

    Returns:
    Returns a list with information about the poll just created.
    See get_poll_return_list for its format.

    Raises:
    - ValueError if the player is not in the instance.

    ""
    if not arguments[0]:
        raise ValueError('Question cannot be empty')
    size = len(arguments[1])
    if size < 2 or size > 5:
        raise ValueError('Incorrect number of options for poll. ' +
                         'Must be between two and five.')

    poll = Message(parent = instance, sender = player,
                    msg_type = 'poll', recipient = '')
    poll.put()
    arguments.append(poll.key().id())
    poll.content = simplejson.dumps(arguments)
    poll.votes = [0] * size
    poll.open = True
    poll.voters = ['']
    poll.put()
    return get_poll_return_list(poll)
```

---

### 3.2.3 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

³The empty recipient field means that the message can be fetched by any user that has joined the game instance by selecting the category.
applications with the SendMessage block (see Figure 3-5). 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 3-5: The method call block for sending a message to other players using the Game Client component.](image)

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.
3.2.4 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 the Game Client component’s WebServiceError event (shown later in Figure 3-10).

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 cause a server command to complete successfully on the Game Server 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 data refresh. 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.

3.2.5 Registering Commands

Before a custom server module can be accessed with the Game Client 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)

```python
custom_command_dict = {
    # Androids to Androids
    'ata_new_game': ata_commands.new_game_command,
    'ata_submit_card': ata_commands.submit_card_command,
    'ata_end_turn': ata_commands.end_turn_command,

    # Bulls and Cows
    'bac_new_game': bac_commands.new_game_command,
    'bac_guess': bac_commands.guess_command,

    # Amazon
    'amz_keyword_search': amazon_commands.keyword_search_command,
    'amz_isbn_search': amazon_commands.isbn_search_command,

    # Voting
    'vot_cast_vote': voting_commands.cast_vote_command,
    'vot_get_results': voting_commands.get_results_command,
    'vot_new_poll': voting_commands.make_new_poll_command,
    'vot_close_poll': voting_commands.close_poll_command,
    'vot_delete_poll': voting_commands.delete_poll_command,
    'vot_get_poll_info': voting_commands.get_poll_information_command,
    'vot_get_my_polls': voting_commands.get_my_polls_command
}
```

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 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 3-6. 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.

![Diagram of code blocks for creating new voting categories.](image)

Figure 3-6: 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.
Retrieving and Displaying Polls

After a category has been selected, its polls are retrieved from the server. A category is activated by setting the Game Client component’s instance ID to the name of the category. When the Game Client component makes a request, it includes the game ID\(^4\) 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.

Unlike most other components, the properties of the Game Client component 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 3-7 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 3-8. Each of them invokes a ServerCommand to perform the requested action.

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

\(^4\)The game ID is set by the application creator and hard-coded into each application.
Figure 3-7: Event handlers for retrieving the list of polls owned by the user. After selecting a category, the Game Client 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.

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 number\(^5\). Figure 3-9 shows the blocks for selecting a poll in the poll creation 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 Game Client 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

---

\(^5\)This 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 3-8: 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.
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.

To make dealing with these errors less of a burden, the voting server module was designed to handle repeated inputs of the same command from an out-of-sync client. Thus, as can be seen in Figure 3-10, 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 3-10: 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 The Ballot Box

The Mobile Voting Ballot Box 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 3-11. Polls are retrieved using the Game Client component’s built-in GetMessages call. The message type specified in Figure 3-11 is the empty string. This tells the server not to 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 3-12. The 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.

3.4 Summary

The entire Mobile Voting application includes two Android programs and a custom server module. The Android programs were created using App Inventor. First
Figure 3-11: 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.

the application designer was used to select components and lay out the user interface. Then, the program logic, including calls to the Game Server, were defined with the blocks editor.

In the blocks editor, the Game Client component provides method call blocks that utilize the Game Server API to keep track of polls and execute server commands in the custom voting module.

The voting module built for the Game Server defines seven different custom commands which perform database operations and provide poll information to
the voting application. Each server command accepts a list of arguments and returns a results list which can be interpreted by the Game Client component and used to display poll information.

Now that the application creation process has been explained, Chapter 4 will provide an in-depth look at the details of the server API and the blocks that the Game Client component exposes.
Chapter 4

System Overview

This chapter is an overview of the design decisions and implementation of the Building Blocks for Mobile Games multiplayer framework. It first discusses the Game Server request API, extensions, custom server modules, data models and testing strategy. Then, the Game Client 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. The server provides a set of request handlers and server commands which can be called by the Game Client component. Server testing is performed with the NoseGAE plugin for the Nose unit testing system. NoseGAE emulates the App Engine data store on the local file system and runs tests in the restricted App Engine runtime environment[7].

4.1.1 Data Models

The Game Server uses three data models to store game and player information. These data models were presented briefly in Chapter 3.

- Game - Each application has one Game object in the data store. The Game
is identified by the game ID property of the application’s Game Client 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 count.

- **Message** - Message objects are used to send information to a player in a GameInstance. Players must explicitly request messages in order to receive them. Thus, creating a message and storing it in the database is functionally equivalent to sending a message with the Game Client component’s SendMessages block.

Messages can be created in custom server commands or using the SendMessage method call block. Messages contain a 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 Game Client component receives a message it automatically converts the content into YailLists to return to the GotMes-

---

1 This 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 both disallow further players from joining and keep it from appearing in the invited games list of all players.

2 This is similar to the setup used by POP email.

3 Dictionaries 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.
GameInstances and Messages allow dynamic properties to be added to them at runtime. 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 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 committed to the database by calling its "put" method. Later, when the Message is retrieved from the database, all of its dynamic properties can be accessed and modified.

```
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 Game Client 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 Game Client component. Each request handler follows the same execution pattern:

1. The request handler retrieves the POST variables from the Game Client’s request. Every request handler accepts a game ID, instance ID, and player...
ID. The player ID must include the email address of the requesting player. Most requests also require additional variables in order to properly execute. For instance, the SendMessages request handler requires the message type, contents and recipients list to be included in the POST variables.

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 Game Client.

3. The transaction returns the database model specified by the game ID and instance ID of the request along with the result of the request.

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 Game Client up to date on changes to the leadership and membership of the instance.

5. The OperationResponse is converted to JSON and returned to the requester.

   The Game Client component then processes the information in the OperationResponse and triggers appropriate events.

   The nine request handlers and the values they return to the Game Client 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.

   To 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.

   The result of a Game Server transaction is always formatted as a Python dictionary. This dictionary is converted into a JSON object and passed back to the Game Client component in the OperationResponse. If the result contains information such as retrieved messages that must be passed to event handlers, the Game Client component will extract the information from the JSON object and convert it into App Inventor data types.
The Public and Invited lists only include instances that currently have fewer 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 Game Client so that it can use it as a POST variable in future requests.

- **InvitePlayer** - Invites a new player to an instance. Once a player is invited to an instance, the instance ID will appear in his or her 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. When players are invited to a public game, the game’s instance ID will appear 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 is 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 Game Client 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 re-
questing 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 Game Client:
  - Type - The string 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 - All messages returned must have been made after this time. The Game Client component automatically tracks the time stamp of the most recently received Message for each message type. When it makes a GetMessages request, it includes this time stamp in order to ensure that every Message returned by GetMessages has not been previously received.

- 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 string. If multiple recipients are provided, a message is created for each of them. A player may also choose to send an empty recipients 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 requesting player is the leader, then the request succeeds and the email

---

8The Game Client component automatically converts App Inventor lists into JSON when messages are created.
address of the new leader is returned 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 Game Client component so that the game learns of leader changes as soon as possible.

The significance of a leader is determined by the game designer. Some programs, such as Mobile Voting, do not use the leader of their instances and never change them. Other games, such as a card game with a dealer, may pass leadership every turn or at other transition points.

- **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.

Generic handling of server commands allows App Inventor programs to execute user defined procedures on the server without making changes to the Game Client component. This is an important feature because users of App Inventor are not able to make changes to the Game Client 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 of 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 Mobile Voting application. Extensions and custom modules are each explained in more detail below.

In addition to their stated return values, NewInstance, JoinInstance, and LeaveInstance return the same instance lists returned by the GetInstanceLists request. This allows the Game Client 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 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.

- **Card Game** - Deals cards and keeps track of players’ hands for an instance. The deck used can be set to any list of items meant to represent cards, 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 a player’s hand and shuffle the deck. Each time a change is made to a player’s hand, Card Game automatically sends that player’s new hand to him or her as a Message. This allows a player to receive the current state of his or her hand by making a GetMessages request.

These extensions are used in the MoBulls and Cows 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 computer controlled chess player 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 APIs 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. The Mobile Voting application presented in Chapter 3 uses this technique 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. For example, in a lesson about using third-party data providers, a teacher could build a server module that accessed eBay auction listings based on keywords. Students could then be challenged to build applications that used the auction listings in an interesting way, but not have to waste time learning the technical details of connecting a mobile phone to eBay’s services.

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 pro-
vided 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
parameters to pass to the function that implements the server command. The ex-
pected order and makeup of the arguments parameter varies across server com-
mands.

The identical, three-item method signature is required for server commands so
that the ServerCommand request handler can successfully execute the commands
without knowing the expected format of the arguments list.

Custom modules are enabled on a Game Server 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 com-
mand 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
A.9.

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 database on the local machine9. Combined with WebTest,
  NoseGAE allows unit tests to simulate POST requests as they would come

9The mock database is stored in a temporary file on the local hard disk and implements the same
semantics and operations as the App Engine database.
from the Game Client 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 functions for Java annotations which label each function as a property, event handler or method call.

The Game Client component is implemented to interface with the Game Server. The code for the Game Client component and its utility classes is available in Appendix B.

4.2.1 Properties

The Game Client 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 to game creators that each Game Client 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 because changing the instance ID requires successful
completion of the JoinInstance server request and cannot simply be changed in 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 locally as a side effect of making any request.

- Players - The list of players that have joined the current instance and not yet left. Note that players do 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 so that changes to the game membership can be disseminated quickly to all players without forcing them to make special requests.

- ServerUrl - The web address of the Game Server. Like the game ID, this can only be set in the component properties panel in the application designer. This is 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 situations\(^\text{10}\). If the setter is used, the UserEmailAddress should be set when a program first opens because the Game Client component has not been designed to handle changes to the UserEmailAddress during program operation. Additionally, allowing play-

\(^{10}\text{This could be necessary if an emulator fails to retrieve a registered email address or an application creator needs to use the same device to simulate multiple players.}\)
ers to set their own email address could result in players spoofing their identity and interfering with games.

4.2.2 Method Calls

The Game Client 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 Game Client 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 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 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 triggered. 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 to implement a message reading loop as seen in the Bulletin Board example in Section 5.1.

4.2.3 Events

Events are triggered automatically by the Game Client 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 Game Client 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** - Triggered automatically at program startup. This should not be used in the Game Client except to set the UserEmailAddress when testing or debugging.

- **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 of a player (including the current one) calling SetLeader or a ServerCommand changing the leader field of the current GameInstance object.

- **NewInstanceId** - 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. His or her 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. The Game Client 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 Game Client component triggers two different error events:
  - **Info** - Triggered when a player attempts to perform an action with improper arguments.
  - **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.

4.3 Summary

Together, the Game Client and Game Server enable a wide variety of applications and games to be built. The next chapter presents four example applications to demonstrate the many different uses of the Building Blocks for Mobile Games multiplayer framework.
Chapter 5

Further Examples

This chapter presents four example programs that make use of the Building Blocks for Mobile Games multiplayer 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 multiuser online message board. Users can create or join different boards and leave messages for others to see.

The second application is a reimplementation of a program made for a class being taught at the University of San Francisco\(^1\). With a custom server module, it accesses Amazon’s E-Commerce Services to look up books by keyword or ISBN. With Building Blocks for Mobile Games, 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 Cows\(^2\) 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.

---

\(^1\)The class, CS 107/103 - “Computing, Robots, and the Web” is being offered during the 2010 spring semester and is taught by Professor David Wolber. Professor Wolber also participated in the original App Inventor pilot program in fall 2009.

\(^2\)Bulls 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.
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 program 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.

Together, these examples illustrate the range of use cases from using an unmodified server, to accessing web services with third party libraries to leveraging server extensions and even modifying database models to fit specific needs.

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. The Bulletin Board design can also be reused by other programs to easily add real time chat capabilities.

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, 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-reading loop and GotMessage handler (shown in Figure 5-3) rely on the assumption that the Game Client 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” 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 96\(^3\). 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

\(^3\)This starting score is chosen so that a guess with zero “bulls” and “cows” on every turn would result in a score of zero.
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.

solution. MoBulls and Cows uses ListPicker components that change their background colors depending on the colors 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 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.

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 players guess 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.
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.

of all players in the game.

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 extra lines of Python code in the server module.

Four of the six lines are shown in Code List 5.1. Two more are required in the New Game command to initialize the dynamic properties to empty values.
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 the server.

5.1: The MoBulls and Cows server command to submit a new guess. Input validation and game ending code has been omitted for brevity. At the end of the method, the last guess and reply are saved as dynamic properties of the Message object that stores the MoBulls and Cows game information. If a subsequent guess has the exact same arguments, the computation and score deduction are skipped. Instead, the saved reply is immediately returned. (Excerpt from A.12)

```
3 def guess_command(instance, player, arguments):
4     guess = arguments[1]
5     game = db.get(Key.from_path('Message', int(arguments[0]),
6     parent = instance.key()))
7     # Check to see if the received guess is identical to the last
```
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.

```python
9  # one received. If so, return the saved reply.
10 if guess == game.bac_last_guess:
11    return simplejson.loads(game.bac_last_reply)
12
13 return_content = None
14
15 if guess == game.bac_solution:
16    # The player has won.
17    # Code omitted for brevity.
18 else:
19    game.bac_guesses_remaining -= 1
20    bulls = cows = 0
21    for i in xrange(solution_size):
22        if guess[i] == game.bac_solution[i]:
23            bulls += 1
24        elif guess[i] in game.bac_solution:
25            cows += 1
26
27    score_deduction = solution_size * 2 - cows - 2 * bulls
28    game.bac_score -= score_deduction
29    return_content = [game.bac_guesses_remaining, game.bac_score,
30                        bulls, cows]
31
32    # Save the guess and reply with the Message object.
33    game.bac_last_reply = simplejson.dumps(return_content)
34    game.bac_last_guess = guess
35    game.put()
```
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 that accesses the Amazon E-Commerce Services API to perform a search of Amazon.com's book inventory.
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.
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 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).

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

```python
def amazon_by_keyword(keyword):
    """Use the ecs library to search for books by keyword."

    Returns:
    A list of three-item lists. Each sublist represents
    a result and includes the book title, its lowest found
    price and its ASIN number.
    ""
    ecs.setLicenseKey(license_key)
    ecs.setSecretKey(secret_key)
    ecs.setLocale('us')

    books = ecs.ItemSearch(keyword, SearchIndex='Books', ResponseGroup='Medium')
    return format_output(books)

def format_output(books):
    """Return a formatted output list from an iterator returned
    by the ecs library. ""

    size = min(len(books), return_limit)
    return [[books[i].Title, get_amount(books[i]), books[i].ASIN]
            for i in xrange(size)]

def get_amount(book):
    """Return the lowest price found or 'Not found.' if none exists.""
    try:
        else:
            return 'Not found.'
    except:
```

83
5.4 Androids to Androids

Androids to Androids is a multiplayer card game played by groups of three or more players\(^5\). 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.

\(^5\)Androids to Androids is a variant of the party game Apples to Apples published by Mattel.
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 criteria he or she 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.

An example round with Bob, Alice and George would follow this rough script:

1. Bob has won the previous round and thus is the current leader.

2. All players receive the adjective card, “closed”.

3. Alice submits the noun card, “apple”.

4. George submits the noun card, “broken”.

5. The leader refreshes his game state and receives the cards that Alice and George submitted. He decides to choose the “apple” card as the winner and submits it to the server.

6. The server looks up the “apple” card and finds out that it was submitted by Alice.

7. Alice receives a point and becomes the leader for the next round. All players are informed that Alice has won the round and are presented with a new adjective card.

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 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.

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

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

arguments: A two-item list consisting of the round to submit this card for and the card itself.

If the submission is for the wrong round, a four-item list with an error string as its first element will be returned. The remaining elements are the player's hand, the current round and the current characteristic card to respond to. No other action will be taken.

Removes the indicated card from the player's hand and adds it to this round's submissions. The current submissions are sent via message to all players.

The requesting player's hand will be dealt another card after removing the submitted one. The updated hand will be sent to the requesting player in a message and be included in the return value of this command.

Returns:
If the submission is for the correct round, returns a three-item
list consisting of the current round number, a list of the
submissions made so far by other players in this round and the
player’s new hand.

Raises:
ValueError if player is the leader. The leader is not allowed to
submit cards.

""
if int(arguments[0]) != instance.ata_round:
    hand = card_game.get_player_hand(instance, player)
    return ['You tried to submit a card for the wrong round. ' +
            'Please try again.\n', hand, instance.ata_round,
            instance.ata_char_card]

if player == instance.leader:
    raise ValueError("The leader may not submit a card.")

submission = arguments[1]
submissions = set_submission(instance, player, submission).values()
instance.create_message(player, 'ata_submissions', '',
                        [instance.ata_round,
                        submissions, submission]).put()

""

5.4.2 Leaders

Andromeda Andromeda uses a game leader to control the user interface and pass leadership of the game from player to player. Each time a request is made, the server includes the current leader in its reply. On the client, every response is checked and a NewLeader event is raised if the value of the leader parameter changes. The NewLeader event handler in Andromeda Andromeda (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.
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.
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 messages 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 his or her 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...
Figure 5-12: The GotMessage event handler in Androids to Androids. Depending on the type of 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.
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 Client component 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.

5.5 Summary

Each of the examples leverages the same set of Game Server capabilities to achieve different outcomes. All use some basic operations such as creating, joining and leaving instances, sending and fetching messages, changing the leader, retrieving instance lists and inviting players. The Bulletin Board application implements a multiuser chatting program without using a single server command or modifying the server in any way. Other programs, such as Amazon or MoBulls and Cows demonstrate that even with a few dozen lines of code, extra capabilities can be added to the Game Server by using custom modules. Finally, we see that with a bit more sophistication, but still using less than 150 lines of code, custom modules can implement more complicated games such as Androids to Androids.
Chapter 6

Related Research

Building Blocks for Mobile Games extends previous work in the fields of visual programming languages, game building, teaching with computer games, and multiuser mobile phone networks. This chapter discusses the design trade-offs of related projects as they compare to the implemented characteristics of the Building Blocks for Mobile Games 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 with graphical building blocks instead of textual code. App Inventor is unique among these languages in its targeting of the mobile phone platform and the breadth of its capabilities.

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
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.

Logo programming language (a dialect of Lisp developed in the late 1960's).

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 Game Client component into a higher tier of complexity. However, Building Blocks for Mobile Games remains accessible to beginning users by simplifying its server requests into single blocks which are easy for application creators to understand.

StarLogoTNG, along with Kodu\(^1\) and Alice\(^2\), 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 proce-

---

\(^1\)Kodu is being developed by a team at Microsoft Research. It was initially released on June 30th 2009 for the Xbox 360 and is currently available for Windows through an invitation only system[3].

\(^2\)Alice was started by Carnegie Mellon University. Electronic Arts and Sun Microsystems are providing development support for the most recent release, which is currently in beta and available for Windows, Linux and OS X[14].
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.

dures 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 by focusing on working in the 3D environment.

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
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]
professors from three universities in Taiwan, uses a client-server setup similar to the Building Blocks for Mobile Games' 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, 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 prerequisite topics[9]. Both of these issues can likely be solved with App Inventor and Building Blocks for Mobile Games 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. However, this has not yet been tested in a real classroom environment.

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 Game Server 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 increased 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 smartphones (and all Android mobile phones, which App Inventor is built for) are capable of using 3rd generation data networks and are generally sold with affordable, unlimited use data plans. Thus, these concerns have become outdated.

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 require state that is not exposed to the user as properties. In the Game Client 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 ListPicker would all make dealing with game interruptions much easier.

One way to solve this is to create a way for component creators to register state variables with some kind of persistence manager. Then, whenever the program is interrupted or closed, all of the registered state is written to the SQLite database running on the phone.

7.3 Pushing Messages and Game State Updates

In the present design of the Game Server, 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. Building Blocks for Mobile Games 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 Building Blocks for Mobile Games multiplayer framework provided the following contributions:

1. Integrated a Game Client 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 the breadth of capabilities of the Building Blocks for Mobile Games 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 Game Client 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 App Engine. All unit tests and third-party code are omitted. For the complete runnable server code, see the App Inventor for Android project on Google Code at:
http://code.google.com/p/app-inventor-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

A1: server.py - The Game Server application file. Includes the request handlers for server requests.

```python
# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
```
The request handlers for the game server. After retrieving arguments from the request, all operations are run as database transactions. This means that any unhandled errors encountered during the operations will result in the database performing a 'rollback' to the state that it was in before the request was made.

All server command functions return a tuple of the database model they operated on and a dictionary of results. These vary from command to command, but all requests will provide their return value using an OperationResponse object.

The get functions for each request handler provide a simple web form to perform the operation via a web interface and will write their responses as a web page. Put functions write json to the request handler that can be consumed by other applications.

Throughout this module, pid is accepted as an argument. The correct format for a pid is of one of the following forms:
'Bill Magnuson' <billmag@mit.edu>
billmag@mit.edu

Received pids will be parsed for the email address and only the email address will be used to identify players during game operations. These same rules apply to other fields which identify players such as a new leader or an invitee. In general, the variable name 'player' will be used to represent values that are email addresses and pid is used more generally to indicate that other strings are acceptable as input.

For more information about the validation done on game ids, instance ids, and player ids, look to utils.py.

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

import sys
import logging
import traceback
import iso8601
import utils
from datetime import datetime
from django.utils import simplejson
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from models.game import Game
from models.game_instance import GameInstance
from models.message import Message
from server_commands import command_dict
# Module Constants
REQUEST_TYPE_KEY = 'request_type'
ERROR_KEY = 'e'
RESPONSE_KEY = 'response'
GAME_ID_KEY = 'gid'
INSTANCE_ID_KEY = 'iid'
PLAYERS_KEY = 'players'
LEADER_KEY = 'leader'

# Request Parameter Keys
PLAYER_ID_KEY = 'pid'
INVITEE_KEY = 'inv'
TYPE_KEY = 'type'
CONTENTS_KEY = 'contents'
COMMAND_KEY = 'command'
ARGS_KEY = 'args'
MESSAGE_COUNT_KEY = 'count'
MESSAGE_RECIPIENTS_KEY = 'mrec'
MESSAGE_TIME_KEY = 'mtime'
INSTANCE_PUBLIC_KEY = 'makepublic'

# Response Helpers

def run_with_response_as_transaction(req_handler, operation, *args, **kwargs):
    """Run operation in a transaction and write its response to req_handler.
    Args:
    req_handler: The request handler to write a response to.
    operation: The callable function to run as a transaction.
    args: Positional arguments to pass to operation.
    kwargs: Keyword arguments to pass to operation.
    Runs operation as a database transaction, creates an OperationResponse with the return value and writes it to the request handler.
    If an exception raises to this function a traceback is written to the debug log and an OperationResponse is written to the request handler with the error message as its contents and the error key set to True.
    """
    try:
        response = db.run_in_transaction(operation, *args, **kwargs)
        OperationResponse(response = response).write_to_handler(req_handler)
    except BaseException, e:
logging.debug('exception encountered: %s' % traceback.format_exc())

OperationResponse(response = e.__str__(), error = True).write_to_handler(req_handler)

class OperationResponse():
    """ Class for handling server operation responses and writing output. """

    An OperationResponse is a standard way to provide a response to a server request. When operations are specific to a game instance, the operation response includes information about the current state of that instance.

    If an error is encountered during an operation the OperationResponse includes only the error boolean and the error's message as its response.

    Attributes:
    error: A boolean indicating that an error occurred during the execution of this operation.
    gid: The game id of the game for this operation.
    iid: The instance id of the game instance.
    leader: The current leader of the game instance.
    players: A list of players in the game instance
    """

def __init__(self, response, error=False):
    """ Fill in parameters based on the error value and the model returned.
    """
    Args:
    response: If no error occurs, response should be a tuple of the database model that this operation was performed with and a dictionary representing the response value of the operation. If an error is encountered, response should be an error message.
    error: A boolean indicating whether the operation encountered an error during execution.

    The OperationResponse's attributes are automatically filled in by reading the attributes of the model in the response tuple. If the model is a Game object then iid, leader and players are left with empty values.

    """
    self.error = error
    self.iid = ''
    self.leader = ''
    self.gid = ''
    self.players = []

    if self.error:
        self.response = response
    else:
        model, self.response = response
if model and model.__class__.__name__ == 'GameInstance':
    self.gid = model.parent().key().name()
    self.iid = model.key().name()
    self.leader = model.leader
    self.players = model.players
elif model and model.__class__.__name__ == 'Game':
    self.gid = model.key().name()

def write_to_handler(self, req_handler):
    """ Writes a response to the req_handler.
    """
    req_handler: The request handler for this server request.

    If the 'fmt' field of this request is 'html' then the response is
    formatted to be written to the web. Otherwise, it is formatted to
    be sent as json.
    ""
    if req_handler.request.get('fmt') == 'html':
        self.write_response_to_web(req_handler)
    else:
        self.write_response_to_phone(req_handler)

def write_response_to_web(self, req_handler):
    """ Writes the response object to the request handler as html.
    """
    req_handler: The request handler for this server request.

    Writes a web page displaying the response object as it would
    be written to json.
    ""
    req_handler.response.headers['Content-Type'] = 'text/html'
    req_handler.response.out.write('<html><body>
    <p>\n    <em>The server will send this to the component:</em>
    ""
    req_handler.response.out.write(''\n    <a href="/\n    <i>Return to Game Server Main Page</i>
    \n    </html>''
    req_handler.response.out.write('</body></html>''

def write_response_to_phone(self, req_handler):
    """ Writes the response object to the request handler as json.
    """
    req_handler: The request handler for this server request.

    ""
    req_handler.response.headers['Content-Type'] = 'application/json'
    req_handler.response.out.write(
        self.get_response_object(req_handler.request.path))
def get_response_object(self, request_type):
    """Return a JSON object as a string with the fields of this response.
    """
    Args:
    request_type: The type of server request that caused this operation.

    Creates a dictionary out of the fields of this object and encodes them in JSON.
    ""
    response = simplejson.dumps({REQUEST_TYPE_KEY : request_type,
                                  ERROR_KEY : self.error,
                                  RESPONSE_KEY : self.response,
                                  GAME_ID_KEY : self.gid,
                                  INSTANCE_ID_KEY : self.iid,
                                  LEADER_KEY : self.leader,
                                  PLAYERS_KEY : self.players})

    logging.debug('response object: %s' % response)
    return response

###########
# Operation Functions #
###########

def get_instance_lists(gid, iid, pid):
    """Return the instances that a player has been invited to and joined.
    """
    Args:
    gid: The game id of the Game object that this method targets.
    iid: The instance id of the Game Instance object that this method targets.
    pid: A string containing the requesting player’s email address.

    The gid and pid must be valid, but the iid can be blank. This is because a player must be able to query for lists of instances without being in one.

    Returns:
    A tuple containing a database model and a dictionary of instance lists. The database model will be a Game Instance if the gid and iid parameters specify a valid GameInstance, otherwise the model will be a Game. Instance lists are returned in the same format as get_instance_lists_dictionary.

    Raises:
    ValueError if the game id or player id are invalid.
    ""
    utils.check_gameid(gid)
    player = utils.check_playerid(pid)
    model = game = utils.get_game_model(gid)
    if game is None:
game = Game(key_name = gid, instance_count = 0)
    model = game
else:
    instance = utils.get_instance_model(gid, iid)
    if instance:
        model = instance
    instance_lists = get_instances_lists_as_dictionary(game, player)
    return model, instance_lists

def invite_player(gid, iid, invitee):
    """Add invitee to the list of players invited to the specified instance.

    Args:
    gid: The game id of the Game object that this method targets.
    iid: The instance id of the Game Instance object that this method targets.
    invitee: The player id of the person to invite.

    Only modifies the instance if the player has not already been invited and has not joined the game.

    Returns:
    A tuple of the game instance and a single item dictionary:
    inv: The email address of the invited player if they are invited. If the player is not invited (because they have already been invited or have already joined the game), the value of 'inv' is the empty string.

    Raises:
    ValueError if the game id, iid or invitee email address are invalid.
    """
    utils.check_gameid(gid)
    utils.check_instanceid(iid)
    player = utils.check_playerid(invitee)
    instance = utils.get_instance_model(gid, iid)
    if player not in instance.invited and player not in instance.players :
        instance.invited.append(player)
        instance.put()
    else:
        player = ''
        return instance, {INVITEE_KEY : player}

def join_instance(gid, iid, pid):
    """Attempt to add a player to an instance.

    Args:
    gid: The game id of the Game object that this method targets.
    iid: The instance id of the Game Instance to join.
    pid: A string containing the requesting player's email address.
A player can join a game instance if it is not full and either the instance is public or the player has been invited. If this operation is invoked by a player that is not current in the specified instance and they are unable to join, it will fail.

If the player is already in the game instance this will succeed without modifying the instance.

If the specified game instance doesn't exist, it will be created as in new_instance with the specified instance id.

If no players are in the game when this player tries to join they will automatically become the leader.

Returns:
A tuple of the game instance and the instance list dictionary for this player (see get_instance_lists_as_dictionary).

Raises:
ValueError if the game id, instance id or player id are invalid.
ValueError if the player is not already in the game and is unable to join.

```
utils.check_gameid(gid)
utils.check_instanceid(iid)
player = utils.check_playerid(pid)
instance = utils.get_instance_model(gid, iid)
if instance is None:
    return new_instance(gid, iid, pid)
if iid in instance_lists['invited']:
    instance_lists['invited'].remove(instance.key().name())
if iid not in instance_lists['joined']:
    instance_lists['joined'].append(instance.key().name())
return instance, instance_lists
```

def leave_instance(gid, iid, pid):
    """ Remove a player from an instance."

    Args:
    gid: The game id of the game object that this method targets.
    iid: The instance id of the Game Instance to remove the player from.
    player: The player wishing to leave the instance.

    If the player that leaves the instance is the leader, the first player on the players lists becomes the leader.

    If no players are left, the maximum number of players allowed in this instance is set to -1 so that no one may join it in the
future. This means that if someone tries to create an instance in
the future with the same instance id, they will end up with one with
a number appended to it (because this GameInstance object will still
exist).

The decision to do this was made because it is not yet possible to
reliably delete all of the messages in a game instance (see
models/game_instance.py). Thus, if players are able to join an
orphaned instances, the old messages could still be available. If,
in the future, App Engine adds ways to reliably delete database
models this behavior could be changed to delete the instance
entirely if everyone leaves.

Returns:
A tuple of the game object and the instance list dictionary
for this player (see get_instance_lists_as_dictionary).

Raises:
ValueError if the player is not currently in the instance.

utils.check_gameid(gid)
utils.check_instanceid(iid)
instance = utils.get_instance_model(gid, iid)
player = instance.check_player(pid)
instance.players.remove(player)
if player == instance.leader and len(instance.players) != 0:
    instance.leader = instance.players[0]
if len(instance.players) == 0:
    instance.maxplayers = -1
instance_lists = get_instance_lists_as_dictionary(game, player)
instante_lists['joined'].remove(instance.key().name())
instance.put()
return game, instance_lists

def get_messages(gid, iid, message_type, recipient, count, time):
    
    Args:
gid: The game id of the Game object that is a parent of the
desired instance.
iid: This instance id of the Game Instance to fetch messages
from.
message_type: A string 'key' for the message. If message_type is
the empty string, all message types will be returned.
recipient: The player id of the recipient of the messages. This
operation will also return messages that are sent with an empty
recipient field.
count: The maximum number of messages to retrieve.
time: A string representation of the earliest creation time of a
message to returned. Must be in ISO 8601 format to parse
correctly.

    Uses the get_messages function of the GameInstance class to
retrieve messages.

Returns:
   A tuple of the game instance and a dictionary with two items:
   'count': The number of messages returned.
   'messages': A list of the dictionary representations of the
   fetched messages.

"
utils.check_gameid(gid)
utils.check_instanceid(iid)
instance = utils.get_instance_model(gid, iid)
recipient = instance.check_player(recipient)
messages = instance.get_messages(count=count,
                                  message_type=message_type,
                                  recipient=recipient, time=time)
return instance, {MESSAGE_COUNT_KEY : len(messages),
                 'messages' : messages}

def new_instance(gid, iid_prefix, pid, make_public = False):
   ""
   Create a new instance of the specified game.
   
   Args:
      gid: The game id of the Game parent of the new instance.
      iid_prefix: The desired instance id. If no instance has been made
      with this name before, then this will be the instance id of the
      newly created instance. However, since instance ids must be
      unique, the actual instance id will likely be iid_prefix with a
      number suffix.
      pid: The id of the first player and leader of the game.
      make_public: A boolean indicating whether this instance should
      be able to be seen and joined by anyone.
   
   The instance id will start with iid_prefix, but could have any
   suffix. If the parent Game object does not exist, it will
   automatically be created.
   
   Returns:
      A tuple of the newly created instance and an instance lists
      dictionary (see get_instance_lists_as_dictionary).
   
   Raises:
      ValueError if the gameid or player id are invalid.
   ""
utils.check_gameid(gid)
player = utils.check_playerid(pid)
game = Game.get_by_key_name(gid)
if game is None:
game = Game(key_name = gid, instance_count = 0)
if not iid_prefix:
iid_prefix = player + 'instance'
instance = game.get_new_instance(iid_prefix, player)
instance_lists = get_instances_lists_as_dictionary(game, player)
instance_lists['joined'].append(instance.key().name())

if make_public:
    instance.public = True
    instance_lists['public'].append(instance.key().name())
instance.put()
game.put()

return instance, instance_lists

def new_message(gid, iid, pid, message_type, message_recipients, message_content):
    """Create new messages and put them in the database.""
    utils.checkgameid(gid)
    utils.check_instanceid(iid)
    instance = utils.get_instance_model(gid, iid)
    player = instance.check_player(pid)
    recipients_list = None
    if message_recipients != '':
        recipients_list = simplejson.loads(message_recipients)
        if isinstance(recipients_list, basestring):
            recipients_list = [recipients_list]
    for recipient_entry in recipients_list:
        if recipient_entry:
            recipient_entry = instance.check_player(recipient_entry)
        message = Message(parent=instance, sender=player,
            content=simplejson.dumps(message_content),
            message_type=message_type)
        if message_recipients == '':
            message_recipients = []
            message_recipients.append(recipient_entry)
        else:
            message_recipients.append(recipient_entry)
        message.put()
msg_type = message_type,
recipient = recipient_entry,
content = message_content)

message_list.append(message)
db.put(message_list)
return instance, {MESSAGE_COUNT_KEY : len(message_list),
MESSAGE_RECIPIENTS_KEY : recipients_list}

def server_command(gid, iid, pid, command, arguments):
    """Performs the desired server command.

Args:
gid: The game id of the Game model for this operation.
iid: The instance id of the GameInstance model for this operation.
pid: The player id of the requesting player.
command: The key identifying the command to execute.
arguments: JSON representation of arguments to the command.

If the gid and iid specify a valid game instance model it will be passed to the server command. In the case that the iid is empty or refers to a game instance that doesn't exist, a game model will be used. Most commands will fail if passed a game model instead of a game instance, but some are indifferent to the model passed to them.

Unless the dynamic property do_not_put has been set to False, this will put the database model after the command has been performed. This means that server commands do not need to make intermediate puts of the instance model passed to them.

Returns:
A tuple of the model used in the server command's execution and a two item dictionary:
'type': The requested command key.
'contents': A Python value of the response value of the command. This varies among server commands but must always be able to be encoded to JSON.

Raises:
ValueError if the game id or player id is invalid.
ValueError if the arguments json cannot be parsed.
ValueError if command is not a known server command.
""
utils.check_gameid(gid)
player = utils.check_playerid(pid)
model = None
if iid:
    model = utils.get_instance_model(gid, iid)
if model is None:
    model = utils.get_game_model(gid)
if model is None:
    model = Game(key_name = gid, instance_count = 0)
arguments = simplejson.loads(arguments)
reply = ''

if command in command_dict:
    reply = command_dict[command](model, player, arguments)
if 'do_not_put' not in model.dynamic_properties() or not model.
do_not_put:
    model.put()
else:
    raise ValueError("Invalid server command: %s." % command)
if not isinstance(reply, list):
    reply = [reply]
return model, {TYPE_KEY : command, CONTENTS_KEY: reply}

def set_leader(gid, iid, pid, leader):
    """Set the leader of the specified instance.

    Args:
    gid: The game id of the GameInstance object's parent Game object.
    iid: The instance id of the GameInstance to change the leader of.
    pid: The player id of the requesting player. This player must be
        the current instance leader in order to change the leader value.
    leader: The player id of the new leader.

    Returns:
    A tuple of the change game instance model and a dictionary with
    two items:
        'current_leader' : The leader after attempting this change.
        'leader_changed' : Whether or not this attempt to set the leader
        succeeded.
    Raises:
    ValueError if the game id or instance id are invalid.
    ValueError if player or leader are not in the specified game
    instance.
    ""
    utils.check_gameid(gid)
    utils.check_instanceid(iid)
    instance = utils.get_instance_model(gid, iid)
    player = instance.check_player(pid)
    leader = instance.check_player(leader)
    if player != instance.leader or instance.leader == leader:
        return instance, {'current_leader' : instance.leader,
            'leader_changed' : False}
    instance.leader = leader
    instance.put()
    return instance, {'current_leader' : leader,
            'leader_changed' : True}

def get_instance(gid, iid):
    """Retrieves an instance and its dictionary.

    Args:
    gid: The game id of the desired GameInstance object's parent Game
object.
iid: The instance id of the desired GameInstance object.

Returns:
A tuple of the game instance object and its dictionary representation.

Raises:
ValueError if the game id or instance id are not valid.

utils.check_gameid(gid)
utils.check_instanceid(iid)
instance = utils.get_instance_model(gid, iid)
return instance, instance.to_dictionary()

###########
# Writer Helpers #
###########

def get_instances_lists_as_dictionary(game, player):
    """ Return a dictionary with joined and invited instance id lists for player.
    """
    Args:
    game: The Game database model that is the parent of the instances to query.
    player: The email address of the player to get instance lists for.
    
    Returns:
    A dictionary of lists:
    'joined' : The list of instance ids of all instances that the player has joined and not subsequently left.
    'invited' : The list of instance ids of all instances that the player has been invited to and not yet joined.
    """
    return {'joined' : get_instances_joined(game, player),
            'invited' : get_instances_invited(game, player),
            'public' : get_public_instances(game)}

def get_instances_joined(game, player):
    """ Return the instance ids of instance that player has joined.
    """
    Args:
    game: The parent Game database model to query for instances.
    player: The email address of the player to look for in instances.
    
    Returns:
    An empty list if game is None. Else, returns a list of the instance ids of all instances with game as their parent that have player in their joined list.
    """
    if game is None:
        return []
query = game.get_joined_instance_keys_query(player)
return [key.name() for key in query]
def get_instances_invited(game, player):
    """ Return the instance ids of instances that player has been
    invited to.
    """
    Args:
    game: The parent Game database model to query for instances.
    player: The email address of the player to look for in instances.
    Returns:
    An empty list if game is None. Else, returns a list of the
    instance ids of all instances with game as their parent that have
    player in their invited list.
    """
    if game is None:
        return []
    query = game.get_invited_instance_keys_query(player)
    return [key.name() for key in query]

def get_public_instances(game):
    """ Return the instance ids of public instances for the specified
    game.
    """
    Args:
    game: The parent Game database model to query for instances.
    Returns:
    An empty list if game is None. Else, returns a list of the
    instance ids of all joinable public instances with game as
    their parent.
    """
    if game is None:
        return []
    query = game.get_public_instances_query(keys_only=True)
    return [key.name() for key in query]

    # Request Handler Classes #

class MainPage(webapp.RequestHandler):
    """ The request handler for the index page of the game server. """
    def get(self):
        """Write a simple web page for displaying server information. """
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write('<html><body>
        <hl>Game Server for App Inventor Game
        " Client Component</hl>
        self.write_game_list()
        self.write_methods()
        self.response.out.write('</p><a href="http://appengine.google.com">')
def write_game_list(self):
    """ Create an HTML table showing game instance information. """
    <p>&lt;table border=1&gt;
    &lt;tr&gt;
        &lt;th&gt;Created &lt;th&gt;Game &lt;th&gt;Instance &lt;th&gt;Players &lt;th&gt;Invitees &lt;th&gt;Leader &lt;th&gt;Public &lt;th&gt;Max Players &lt;th&gt;More ...
        &lt;/tr&gt;
    games = db.GqlQuery("SELECT * FROM GameInstance")
    for game in games:
        self.response.out.write('&lt;tr&gt;&lt;td&gt;\% UTC&lt;/td&gt;\n' % game.date.ctime())
        self.response.out.write('&lt;td&gt;\% &lt;/td&gt;\n' % game.parent().key().name())
        self.response.out.write('&lt;td&gt;\% &lt;/td&gt;\n' % game.key().name())
        for player in game.players:
            self.response.out.write('&lt;td&gt;\% &lt;/td&gt;\n' % player)
        for invite in game.invited:
            self.response.out.write('&lt;td&gt;\n' % invite)
        self.response.out.write('&lt;td&gt;\n' % game.leader)
        self.response.out.write('&lt;td&gt;\n' % game.public)
        self.response.out.write('&lt;td&gt;\n' % game.max_players)
        self.response.out.write('&lt;/td&gt;
' %
        &lt;form action="/getinstance" method="post" %
        enctype=x-www-form-urlencoded&gt;
        &lt;input type="hidden" name="gid" value="\%s"&gt;
        &lt;input type="hidden" name="iid" value="\%s"&gt;
        &lt;input type="hidden" name="fmt" value="html">% % game.parent().key().name(), game.key().name())
        self.response.out.write('&lt;/form&gt;&lt;/td&gt;
' %
        &lt;/tr&gt;')
def write_methods(self):
    """
    Write links to the available server request pages.
    """
    self.response.out.write(''
    <p>Available calls:
    <ul>
    <li><a href="/newinstance">/newinstance</a></li>
    <li><a href="/invite">/invite</a></li>
    <li><a href="/joininstance">/joininstance</a></li>
    <li><a href="/leaveinstance">/leaveinstance</a></li>
    <li><a href="/newmessage">/newmessage</a></li>
    <li><a href="/messages">/messages</a></li>
    <li><a href="/setleader">/setleader</a></li>
    <li><a href="/getinstance">/getinstance</a></li>
    <li><a href="/getinstancelists">/getinstancelists</a></li>
    <li><a href="/servercommand">/servercommand</a></li>
    </ul>''')

class GetInstanceLists(webapp.RequestHandler):
    """
    Request handler for the get_instance_lists operation.
    """
    def post(self):
        """
        Execute get_instance_lists and write the response to the
        handler.
        """
        Request parameters:
        gid: The game id of the parent Game to get instances of.
        iid: The instance id of the game instance to execute the
             command with. This is optional for this command, although,
             including it will result in the ResponseObject including
             leader and player information.
        pid: The player id of the requesting player.
        """
        logging.debug('/getinstancelists?%s
l%s|' %
            (self.request.query_string, self.request.body))
        gid = self.request.get(GAME_ID_KEY)
        iid = self.request.get(INSTANCE_ID_KEY)
        pid = self.request.get(PLAYER_ID_KEY)
        run_with_response_as_transaction(self, get_instance_lists, gid, iid, pid)

    def get(self):
        """
        Write a short HTML form to perform a get_instance_lists
        operation."""
        self.response.out.write(''
            <html><body>
            <form action="/getinstancelists" method="post"
                enctype=application/x-www-form-urlencoded>
                <p>Game ID <input type="text" name="gid" /></p>
                <p>Instance ID <input type="text" name="iid" /></p>
                <p>Player ID <input type="text" name="pid" /></p>
                <input type="hidden" name="fmt" value="html">
                <input type="submit" value="Get Instance Lists">
            </form>''
        </body>''
class GetMessages(webapp.RequestHandler):
    """ Request handler for the get_messages operation. """
    def post(self):
        """ Execute get_messages and write the response to the handler. """

        Request parameters:
        gid: The game id of the parent Game.
        iid: The instance id of the game instance to execute the command with.
        pid: The player id of the message recipient.
        type: The type of messages requested or the empty string to retrieve all messages.
        count: An integer number of messages to retrieve. This is treated as a maximum and defaults to 1000 if there is a failure retrieving the count parameter.
        mtime: A string in ISO 8601 date format. All messages returned will have a creation time later than this time. Defaults to datetime.min if there is a failure in retrieving or parsing the parameter.

    ""
    logging.debug('/messages?%s
l%sl' %
               (self.request.query_string, self.request.body))

    gid = self.request.get(GAME_ID_KEY)
    iid = self.request.get(INSTANCE_ID_KEY)
    message_type = self.request.get(TYPE_KEY)
    recipient = self.request.get(PLAYER_ID_KEY)

    count = 1000
    try:
        count = int(self.request.get(MESSAGE_COUNT_KEY))
    except ValueError:
        pass

    time = datetime.min
    try:
        time_string = self.request.get(MESSAGE_TIME_KEY)
        if time_string is not None and time_string != '':
            time = iso8601.parse_date(time_string)
    except ValueError:
        pass

    run_with_response_as_transaction(self, get_messages, gid, iid,
                                       message_type, recipient, count,
                                       time)

    def get(self):
        """ Write a short HTML form to perform a get_messages operation. """

        self.response.out.write(''
<html><body>
<form action="/messages" method="post"
    encodetype=application/x-www-form-urlencoded>

122
class InvitePlayer(webapp.RequestHandler):
    """ Request handler for the invite_player operation."""
    def post(self):
        """ Execute invite_player and write the response to the handler. """
        logging.debug('/invite?%s
l%sI' %
            (self.request.query_string, self.request.body))
        gid = self.request.get(GAME_ID_KEY)
        iid = self.request.get(INSTANCEIDKEY)
        inv = self.request.get(INVITEEKEY)
        run_with_response_as_transaction(self, invite_player, gid, iid, inv)

def get(self):
    """ Write a short HTML form to perform an invite_player operation. """
    self.response.out.write(''
        <html><body>
        <form action="/invite" method="post"
            enctype=application/x-www-form-urlencoded>
            <p>Game ID <input type="text" name="gid" /></p>
            <p>Instance ID <input type="text" name="iid" /></p>
            <p>Message type <input type="text" name="type" /></p>
            <p>Email <input type="text" name="pid" /></p>
            <p>Count <input type="text" name="count" /></p>
            <p>Time <input type="text" name="mtime" /></p>
            <input type="hidden" name="fmt" value="html">
            <input type="submit" value="Get Messages">
        </form>
        ')

class JoinInstance(webapp.RequestHandler):
    """ Request handler for the join_instance operation."""
    def post(self):
        """ Execute join_instance and write the response to the handler. """
        Request parameters:
        gid: The game id of the parent Game.
        iid: The instance id of the game instance to join.
pid: The player id of the requesting player.

logging.debug('/joininstance?%s
|%s|

(self.request.query_string, self.request.body))
gid = self.request.get(GAME_ID_KEY)
iid = self.request.get(INSTANCE_ID_KEY)
pid = self.request.get(PLAYER_ID_KEY)
run_with_response_as_transaction(self, join_instance, gid, iid, pid)

def get(self):
    """Write a short HTML form to perform a join_instance operation.
    """
    self.response.out.write(''
    <html><body>
    <form action="/joininstance" method="post"
        enctype=application/x-www-form-urlencoded>
        <p>Game ID <input type="text" name="gid" /></p>
        <p>Instance ID <input type="text" name="iid" /></p>
        <p>Player ID <input type="text" name="pid" /></p>
        <input type="hidden" name="fmt" value="html">
        <input type="submit" value="Join Instance">
    </form>''
    self.response.out.write('</body></html>

class LeaveInstance(webapp.RequestHandler):
    """Request handler for the leave_instance operation."""
    def post(self):
        """Execute leave_instance and write the response to the handler.
        """
        Request parameters:
        gid: The game id of the parent Game.
        iid: The instance id of the game instance to leave.
        pid: The player id of the requesting player.
        """
        logging.debug('/leaveinstance?%s
|%s|

(self.request.query_string, self.request.body))
gid = self.request.get(GAME_ID_KEY)
iid = self.request.get(INSTANCE_ID_KEY)
pid = self.request.get(PLAYER_ID_KEY)
run_with_response_as_transaction(self, leave_instance, gid, iid, pid)

def get(self):
    """Write a short HTML form to perform a leave_instance operation.
    """
    self.response.out.write(''
    <html><body>
    <form action="/leaveinstance" method="post"
        enctype=application/x-www-form-urlencoded>
        <p>Game ID <input type="text" name="gid" /></p>
        <p>Instance ID <input type="text" name="iid" /></p>
        <p>Player ID <input type="text" name="pid" /></p>
        <input type="hidden" name="fmt" value="html">
    </form>''
class NewInstance(webapp.RequestHandler):
    """ Request handler for the new_instance operation."""
    def post(self):
        """ Execute new_instance and write the response to the handler. """
        Request parameters:
        gid: The game id of the parent Game.
        iid: The proposed instance id of the new instance. The instance
        id of the created instance could differ from this if the
        proposed id is already in use.
        pid: The player id of the requesting player.
        make_public: A boolean indicating whether this instance should
        be able to be seen and joined by anyone.
        """
        logging.debug('/newinstance?%s
I%sl' %
            (self.request.query_string, self.request.body))
        gid = self.request.get(GAME_IDKEY)
        iid = self.request.get(INSTANCE_IDKEY)
        pid = self.request.get(PLAYER_IDKEY)
        make_public = False
        try:
            make_public = utils.getboolean(self.request.get(INSTANCE_PUBLIC_KEY))
        except ValueError:
            pass
        runwithresponse astransaction(self, new_instance, gid, iid,
                pid, make_public)
    def get(self):
        """ Write a short HTML form to perform a new_instance operation. """
        self.response.out.write(''
            <form action="/newinstance" method="post"
                enctype=application/x-www-form-urlencoded>
                <p>Game ID <input type="text" name="gid" /></p>
                <p>Instance ID <input type="text" name="iid" /></p>
                <p>First player ID <input type="text" name="pid" /></p>
                <input type="hidden" name="fmt" value="html">
                <input type="submit" value="New Instance">
            </form>
        ')

class NewMessage(webapp.RequestHandler):
    """ Request handler for the new_message operation. """
    def post(self):
        """ Execute new_message and write the response to the handler. """
        Request parameters:
        gid: The game id of the parent Game.
iid: The instance id of the game instance add messages to.

pid: The player id of the requesting player.

type: The message type key.

mrec: Json representation of the recipients of the message.

content: Json representation of the contents of the message.

""
logging.debug('/newmessage?%s
I%sI' %
(self.request.query_string, self.request.body))

gid = self.request.get(GAME_ID_KEY)
pid = self.request.get(PLAYER_ID_KEY)
iid = self.request.get(INSTANCE_ID_KEY)
type = self.request.get(TYPE_KEY)
messagerecipients = self.request.get(MESSAGE_RECIPIENTS_KEY)
messagecontent = self.request.get(CONTENTS_KEY)
run_with_response_as_transaction(self, new_message, gid, iid, pid,
message_type, messagerecipients,
messagecontent)

def get(self):
    """ Write a short HTML form to perform a new_message operation."""
    self.response.out.write(''
<html><body>
<form action="/newmessage" method="post"
  enctype=application/x-www-form-urlencoded>
  <p>Game ID <input type="text" name="gid" /></p>
  <p>Instance ID <input type="text" name="iid" /></p>
  <p>Player ID <input type="text" name="pid" /></p>
  <p>Message type <input type="text" name="type" /></p>
  <p>Message Recipients (Json array) <input type="text" name="mrec" /></p>
  <p>Message Contents (Json array) <input type="text" name="contents" /></p>
  <input type="hidden" name="fmt" value="html">
  <input type="submit" value="Send Message">
</form>''

self.response.out.write(''
<p> Expected format for recipients: <br>
["email@domain.com", "email2@domain.com"]
</p>''

self.response.out.write(''
<p> Expected format for contents: <br>
["string 1", "string 2"]</p>''

self.response.out.write(''
</body></html>
''

class ServerCommand(webapp.RequestHandler):
    """ Request handler for the servercommand operation. """

def post(self):
    """ Execute servercommand and write the response to the handler. 

Request parameters:

  gid: The game id of the parent Game to execute the command with."

}
iid: The instance id of the game instance to execute the command with.

pid: The player id of the requesting player.

command: The key of the command.

arguments: Json representation of the arguments to the server command.

logging.debug('/servercommand?%s\n%s' %
             (self.request.query_string, self.request.body))

gid = self.request.get(GAME_ID_KEY)
nid = self.request.get(INSTANCE_ID_KEY)
pid = self.request.get(PLAYER_ID_KEY)
command = self.request.get(COMMAND_KEY)
arguments = self.request.get(ARG_KEY)
run_with_response_as_transaction(self, server_command, gid, nid, pid,
                                 command, arguments)

def get(self):
    """ Write a short HTML form to perform a set_leader operation. """
    self.response.out.write(''
<html><body>
<form action="/servercommand" method="post"
         enctype=application/x-www-form-urlencoded>
<p>Game ID <input type="text" name="gid" /></p>
<p>Instance ID <input type="text" name="iid" /></p>
<p>Player ID <input type="text" name="pid" /></p>
<p>Command <input type="text" name="command" /></p>
<p>Arguments (Json array) <input type="text" name="args" /></p>

<input type="hidden" name="fmt" value="html">
<input type="submit" value="Send Command">
</form>''
self.response.out.write('</body></html>')

class SetLeader(webapp.RequestHandler):
    """ Request handler for the set_leader operation. """
    def post(self):
        """ Execute set_leader and write the response to the handler. """
        Request parameters:
        gid: The game id of the parent Game.
nid: The instance id of the game instance to change the leader of.
leader: The player id of the new leader candidate.
pid: The player id of the requesting player.

        logging.debug('/setleader?%s\n%s' %
                       (self.request.query_string, self.request.body))
gid = self.request.get(GAME_ID_KEY)
nid = self.request.get(INSTANCE_ID_KEY)
leader = self.request.get(LEADER_KEY)
pid = self.request.get(PLAYER_ID_KEY)
run_with_response_as_transaction(self, set_leader, gid, iid, pid, leader)

def get(self):
    r""" Write a short HTML form to perform a set_leader operation."""
    self.response.out.write(''
    <html><body>
    <form action="/setleader" method="post"
        enctype=application/x-www-form-urlencoded>
        <p>Game ID <input type="text" name="gid" /></p>
        <p>Instance ID <input type="text" name="iid" /></p>
        <p>Player ID <input type="text" name="pid" /></p>
        <p>New leader (player id) <input type="text" name="leader" /></p>
    </form>''
    self.response.out.write('</body></html>
')

class GetInstance(webapp.RequestHandler):
    r""" Request handler for the get_instance operation."""
    def post(self):
        r""" Execute get_instance and write the response to the handler.
        Request parameters:
        gid: The game id of the parent Game.
        iid: The instance id of the game instance to get the
             information of.
        """
        logging.debug('/getinstance?%s
l%s|'
            % (self.request.query_string, self.request.body))
        gid = self.request.get(GAME_IDKEY)
        iid = self.request.get(INSTANCEIDKEY)
        run_with_response_as_transaction(self, get_instance, gid, iid)
        self.response.out.write(''
        <html><body>
        <form action="/getinstance" method="post"
            enctype=application/x-www-form-urlencoded>
            <p>Game ID <input type="text" name="gid" /></p>
            <p>Instance ID <input type="text" name="iid" /></p>
            <input type="hidden" name="fmt" value="html">
            <input type="submit" value="Get Instance Info">
        </form>''
        self.response.out.write('</body></html>
')

##########################

# Handlers not used by GameClient component #

class GetInstance(webapp.RequestHandler):
    r""" Request handler for the get_instance operation."""
    def post(self):
        r""" Execute get_instance and write the response to the handler.
        Request parameters:
        gid: The game id of the parent Game.
        iid: The instance id of the game instance to get the
             information of.
        """
        logging.debug('/getinstance?%s
l%s|'
            % (self.request.query_string, self.request.body))
        gid = self.request.get(GAME_IDKEY)
        iid = self.request.get(INSTANCEIDKEY)
        run_with_response_as_transaction(self, get_instance, gid, iid)
        self.response.out.write(''
        <html><body>
        <form action="/getinstance" method="post"
            enctype=application/x-www-form-urlencoded>
            <p>Game ID <input type="text" name="gid" /></p>
            <p>Instance ID <input type="text" name="iid" /></p>
            <input type="hidden" name="fmt" value="html">
            <input type="submit" value="Get Instance Info">
        </form>''
        self.response.out.write('</body></html>
')

##########################
def application(custom_command_dict):
    """Return the WSGI Application with the game server request handlers.

    Args:
    custom_command_dict: A dictionary of command name strings to functions.

    The custom_command_dict will be added to the server's command
dictionary so that custom commands can be invoked with the
ServerCommand request handler. If command names in
custom_command_dict are the same as built in server commands they
will overwrite the built in functions.
    """
    for command in custom_command_dict.iteritems():
        command_dict[command[0]] = command[1]
    return webapp.WSGIApplication([('/','MainPage),
        ('/newinstance', NewInstance),
        ('/invite', InvitePlayer),
        ('/joininstance', JoinInstance),
        ('/leaveinstance', LeaveInstance),
        ('/newmessage', NewMessage),
        ('/getinstance', GetInstance),
        ('/getinstanceLists'),
        ('/messages', GetMessages),
        ('/setleader', SetLeader),
        ('/servercommand', ServerCommand),
        ('/getinstance', GetInstance)],
        debug=True)
specific server commands are in the extensions folder.

To enable server commands, they must be entered into commands_dict. Every server command should take in a database model (either a game instance or a game), the email address of the player that requested the server command and a list of arguments. The format of the arguments and what is done with the player and the database model depends on the command.

Server commands should be generally useful to creators of games and not be created for a specific game. Additionally, command functions should, where appropriate, only parse the arguments and perform their actual operations in separate methods. This allows for custom modules to utilize extensions more easily when creating game specific functions.

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>]}

import logging
import traceback
import utils
from google.appengine.api import mail
from google.appengine.ext import db
from google.appengine.runtime import apiproxy_errors
from django.utils import simplejson
from extensions import scoreboard
from extensions import card_game

EMAILSENDER = ""

def send_email_command(model, player, arguments):
    ""
    Send an email using App Engine's email server.
    Arg:
    instance: Not used, can be any value.
    player: The player requesting that the email be sent.
    arguments: A two item list with the subject of the email as the first item and the body of the email as the second.
    EMAILSENDER must be defined above and be a listed developer of your AppEngine application for this to work successfully.
    Returns:
    True if the email sends successfully, False otherwise.
    ""
    if email_sender:
        message_recipient = arguments[0]
        message_content = arguments[1]
        mail.send_mail(sender=EMAILSENDER,
                        to=message_recipient,
                        subject=message_content[0],
                        body=message_content[1] +
                        '
Message sent from AppInventorGameServer by' +
def set_public_command(instance, player, arguments):
    """ Set the public membership field for an instance.
    
    Args:
    instance: The GameInstance database model to change.
    player: The player requesting the change. This player must be the current leader of the instance.
    arguments: A single item list containing the desired boolean value for the public field of instance.
    
    A public game can be joined by players without first being invited. Changing the value of public does not change the current membership of the game.
    
    Returns:
    The new value of public for the instance.
    
    Raises:
    ValueError if the requesting player is not the leader of the instance.
    ValueError if the argument is unable to be parsed into a boolean.
    """
    instance.check_leader(player)
    value = utils.getboolean(arguments[0])
    instance.public = value
    return value

def set_max_players_command(instance, player, arguments):
    """ Set the maximum number of players allowed to join an instance.
    
    Args:
    instance: The GameInstance database model to change.
    player: The player requesting the change. This player must be the current leader of the instance.
    arguments: A single item list containing the desired integer value for the max players of this instance.
    
    If the maximum player count is set to a value lower than the current number of players, no players will be removed. However, new players will not be able to join until the max players count goes up or enough players leave the instance that the number of players is less than the maximum.
    
    Returns:
    The new value of max_players for the instance.
    
    Raises:
    ValueError if the requesting player is not the leader of the instance.
    ValueError if the argument is unable to be parsed into an integer.
instance.check_leader(player)
max_players = int(arguments[0])
instance.max_players = max_players
return max_players

def get_public_instances_command(model, player, arguments = None):
    """Return a list of public instances of the specified game.
    ""
    Args:
    model: Either a Game or GameInstance database model. If model is a
    GameInstance, this will return the public instances of its
    parent Game.
    player: Not used. Value can be anything.
    arguments: Not used, can be any value.
    Returns:
    A list of all public instances in this game that have less players
    than their maximum (i.e. can be joined). Instances are sorted with
    the newest ones first. Each entry in the list of instances is
    itself a three item list with the instance id as the first item,
    the number of players currently in the game as the second item and
    the maximum number of players (if any) as the third item. If no
    maximum number of players is set for the game instance the third
    item will be set to zero.
    ""
    game = utils.get_game(model)
    public_instances = game.get_public_instances_query().fetch(1000)
    return [[i.key().name(), len(i.players), i.max_players]
             for i in public_instances]

def delete_instance_command(instance, player, arguments = None):
    """Delete an instance and its messages.
    ""
    Args:
    instance: The instance to delete.
    player: The player requesting the deletion. This player must
    be the current leader of the instance.
    arguments: Not used, can be any value.
    Makes a good faith effort to delete the messages, but deleting large
    numbers of database entries is currently very buggy in
    AppEngine. This will hopefully get better over time as AppEngine
    advances. See the method delete_messages in models/game_instance.py
    for more information.
    If the deletion of messages fails the exception will be logged and
    this command will return normally.
    Returns:
    True if the instance deletes succesfully.
    Raises:
    ValueError if player is not the leader of the instance.
if instance.__class__.__name__ != 'GameInstance':
    raise ValueError("Only models of type GameInstance may be deleted. ")

instance.check_leader(player)
try:
    instance.delete_messages()
except apiproxy_errors.ApplicationError, err:
    logging.debug("Exception during message deletion: $s" %
                  traceback.format_exc())
db.delete(instance)
instance.do_not_put = True
return True

def decline_invite_command(instance, player, arguments = None):
    """Remove a player from the invited list of an instance.
    Args:
    instance: The instance to uninvite player from.
    player: The player wishing to decline an invite.
    arguments: Not used, can be any value.
    Returns:
    True if the player was previously invited to the game, False
    otherwise.
    """
    if player in instance.invited:
        instance.invited.remove(player)
    return True
return False

command_dict = {
'sys_email' : send_email_command,
'sys_set_public' : set_public_command,
'sys_set_max_players' : set_max_players_command,
'sys_get_public_instances' : get_public_instances_command,
'sys_delete_instance' : delete_instance_command,
'sys_decline_invite' : decline_invite_command,
# Scoreboard commands.
'scb_get_scoreboard' : scoreboard.get_scoreboard_command,
'scb_get_score' : scoreboard.get_score_command,
'scb_add_to_score' : scoreboard.add_to_score_command,
'scb_set_score' : scoreboard.set_score_command,
'scb_clear_scoreboard' : scoreboard.clear_scoreboard_command,
# Card commands.
'crd_set_deck' : card_game.set_deck_command,
'crd_deal_cards' : card_game.deal_cards_command,
'crd_draw_cards': card_game.draw_cards_command,
'crd_discard': card_game.discard_command,
'crd_pass_cards': card_game.pass_cards_to_player_command,
'crd_cards_left': card_game.get_cards_remaining_command
}

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

# Copyright 2010 Google Inc.

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
# See the License for the specific language governing permissions and
limitations under the License.

""
Utility functions for input validation and database model access.
""

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

import re
from google.appengine.ext import db
from google.appengine.ext.db import Key

EMAIL_ADDRESS_REGEX = ("([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@" + 
"([-0-9a-zA-Z]+[.]|[.]a-zA-Z[0-9a-zA-Z]{2,6})")

def get_game_model(gid):
    """Return a Game model for the given game id.
    Args:
    gid: The game id of the Game.
    Returns:
    The database model for the specified id or None if no such model
    exists.
    """
    game_key = Key.from_path('Game', gid)
    model = db.get(game_key)
    return model

def get_instance_model(gid, iid):
    """Return a GameInstance model for the given game and instance ids.
    Args:
    gid:
    iid:
    """

gid: The game id of the GameInstance.

iid: The instance id of the GameInstance.

Returns: The database model for the specified ids or None if the GameInstance doesn't exist.

""
instance_key = Key.from_path('Game', gid, 'GameInstance', iid)
model = db.get(instance_key)
return model

def check_playerid(pid, instance = None):
    """Return a valid player id.
    ""
    Args:
    pid: A string containing the email address of the player or the special identified 'leader'.
    instance: (optional) The instance from which to fetch the leader from when pid is 'leader'.

    Returns:
    Strips the supplied player id of superfluous characters and returns only the email address. Also does conversion of the special string 'leader' to the current leader of instance.

    Raises:
    ValueError if pid does not match an email address regular expression.
    ""
    if instance and pid.lower() == 'leader':
        pid = instance.leader
    if pid is None or pid == '':
        raise ValueError('The player identifier is blank.')
    stripped_email = re.search(EMAIL_ADDRESS_REGEX, pid)
    if stripped_email is None:
        raise ValueError('%s is not a valid email address.' % pid)
    return stripped_email.group(0)

def check_gameid(gid):
    """Validate the game id to make sure it is not empty.
    ""
    Args:
    gid: The game id to check

    Returns:
    The game id.

    Raises:
    ValueError if the game id is the empty string or None.
    ""
    if gid == '' or gid is None:
        raise ValueError('Bad Game Id: %s' % gid)
    return gid
def check_instanceid(iid):
    """Validate the instance id to make sure it is not empty.
    Args:
    iid: The instance id to check
    Returns:
    The instance id.
    Raises:
    ValueError if the instance id is the empty string or None.
    """
    if iid == """ or iid is None:
        raise ValueError('No instance specified for request.')
    return iid

def get_boolean(value):
    """Return a bool from value.
    Args:
    value: A string or bool representing a boolean.
    Returns:
    If value is a bool, value is returned without modification.
    If value is a string this will convert 'true' and 'false'
    (ignoring case) to their associated values.
    Raises:
    ValueError if value does not match one of the string tests and is
    not a bool.
    """
    if type(value) is not bool:
        value = value.lower()
        if value == 'true':
            value = True
        elif value == 'false':
            value = False
        else:
            raise ValueError("Boolean value was not valid")
    return value

def get_game(model):
    """Return a Game object.
    Args:
    model: A database model that is either a GameInstance or Game.
    Returns:
    Either returns model or its parent if either of them is a Game
    object.
    Raises:
ValueError if either model or its parent is not a Game object.

if model._class__.__name__ == 'GameInstance':
    model = model.parent()
if model._class__.__name__ == 'Game':
    return model
raise ValueError('Invalid model passed to get_game')

A.1.1 Models

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

# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

from google.appengine.ext import db
from game_instance import GameInstance

class Game(db.Model):
    """A model for a game type.
    A Game is the parent object of GameInstance objects. Each Game
    object should correspond to a type of game. Class methods are
    available that provide queries to discover GameInstance objects
    with this parent.
    The key_name of a Game should be the game's name.
    Attributes:
    instance_count: The number of instances that have been made with
    this Game as their parent. This number is managed manually.
    ""
    instance_count = db.IntegerProperty(default=0)

    def get_new_instance(self, prefix, player):
        """Create a new GameInstance and return its model."""
Args:
  prefix: A string used as the beginning of the instance id.
  player: The email address of the player.

When this returns, neither this Game model or the new
GameInstance have been put() in the database. If the GameInstance
should persist, both models need to be put() in.

Returns:
  A GameInstance object with a unique instance id beginning with
  prefix and player as the leader and sole member of the
  instance.

"""
prefix = prefix.replace(' ', '')
new_iid = prefix
self.instance_count += 1
new_index = self.instance_count
while GameInstance.get_by_key_name(new_iid, parent=self) is not None:
    new_index += 1
    new_iid = prefix + str(new_index)
instance = GameInstance(parent=self, key_name=new_iid,
    players=[player], leader=player)
return instance
"""

def get_public_instances_query(self, keys_only=False):
    """ Return a query object for public instances of this game.
    """
    Args:
        keys_only (optional): Whether this database query should return
        only keys, or entire models.
    
    Returns:
        A query object of all public game instances that are not full
        in order of creation time from oldest to newest. Any instance
        returned by this query should be able to be joined by any
        player at the time the results are fetched.

    """
    query = GameInstance.all(keys_only=keys_only)
    query.filter("public =", True)
    query.filter("full =", False)
    query.ancestor(self.key())
    query.order('-date')
    return query

def get_invited_instance_keys_query(self, player):
    """ Return a query object for instances a player has been invited to.
    """
    Args:
        player: The email address of the player.
    
    Returns:
A query object of all game instances that player has been invited to and that are not full in order of creation time from oldest to newest. Any instance returned by this query should be able to be joined by the player at the time the results are fetched.

```python
query = GameInstance.all(keys_only = True)
query.filter("invited =", player)
query.filter("full =", False)
query.ancestor(self.key())
query.order('-date')
return query
```

def get_joined_instance_keys_query(self, player):
    """Return a query object for instances a player has already joined.

    Args:
        player: The email address of the player.

    Returns:
        A query object of all game instances that player has joined in order of creation time from oldest to newest.
    """
    query = GameInstance.all(keys_only = True)
    query.filter("players =", player)
    query.ancestor(self.key())
    query.order('-date')
    return query
```

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

```python
__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

from datetime import datetime
from django.utils import simplejson
from gameserver import utils
from google.appengine.ext import db
```

```python
# Copyright 2009 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#     http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
""" The GameInstance database model class and associated methods."""
```
```python
from message import Message

class GameInstance(db.Expando):
    """ A model for an instance of a game.
    """
    # A GameInstance contains all of the membership and message
    # information for an instance of a particular game. It is implemented
    # as an Expando model to allow extensions and custom modules to add
    # dynamic properties in order to extend the functionality of the
    # instance.
    
    # The key_name of a GameInstance should be the unique instance id. A
    # GameInstance's parent is the Game model that it is an instance of.
    Attributes:
        players: A list of the email addresses of players currently in the
                instance.
        invited: A list of email addresses of invited players.
        leader: The player that is currently the leader of the instance.
        date: The date of creation, automatically set upon instantiation.
        public: A bool that determines whether a player must first be
                invited before they can join this instance.
        full: A boolean indicating whether or not the game has reached
              its maximum membership. Automatically set when the GameInstance
              is put.
        max_players: An integer for the maximum number of players allowed
                     in this instance or 0 if there is no maximum.

    """
    players = db.StringListProperty(required=True)
    invited = db.StringListProperty(default=[])
    leader = db.StringProperty(required=True)
    date = db.DateTimeProperty(required=True, auto_now=True)
    public = db.BooleanProperty(default=False)
    full = db.BooleanProperty(default=False)
    max_players = db.IntegerProperty(default=0)

    def put(self):
        """ Set the value of full and put this instance in the database.
        """
        self.set_full()
        db.Model.put(self)

    def set_full(self):
        """ Set the full attribute of this entity appropriately.
        """
        # This should be called at put time to make sure that any stored
        # GameInstance model has the appropriate value for full. A game is
        # full if it has a non-zero value for max players which is less than
        # or equal to the number of players in the game.
        if self.max_players == 0 or self.max_players > len(self.players):
            self.full = False
        else:
            pass
```
self.full = True

def to_dictionary(self):
    """ Return a dictionary representation of the instance's attributes. """
    return {'gameid': self.parent().key().name(),
            'instanceId': self.key().name(),
            'leader': self.leader,
            'players': self.players,
            'invited': self.invited,
            'public': self.public,
            'max_players': self.max_players}  

def __str__(self):
    """ Return a json string of this model's dictionary. """
    return simplejson.dumps(self.to_dictionary())

def create_message(self, sender, msg_type, recipient, content):
    """ Create a new message model with this instance as its parent. 
    Args:
        sender: A string describing the creator of the message.
        msg_type: A string that acts as a key for the message.
        recipient: The intended recipient of this message. The
            recipient should either be the empty string or an email
            address. Messages sent with the empty string as their
            recipient can be fetched by any player.
        content: A python list or dictionary representing the content
            of this message. Converted to a json string for storage.
    Returns:
        A new Message model with the specified attributes. This model
        has not yet been put in the database.
    """
    return Message(parent = self, sender = sender, msg_type = msg_type,
                   recipient = recipient, content = simplejson.dumps(
                   content))

def get_messages(self, time=datetime.min, count=1000,
                 message_type='', recipient ''):
    """ Return a list of message dictionaries using query_messages. 
    Args (optional):
        time: (default: datetime.min) All messages retrieved must have 
            been created after this time.
        count: (default 1000) The maximum number of messages to 
            retrieve.
        message_type: (default '') The message type to retrieve. If
            left as None or the empty string then all messages matching
            other criteria will be returned.
        recipient: (default '') The recipient of the messages to
            retrieve. All messages sent with a recipient of the empty
            string will also be retrieved.
Returns:
The dictionary representation of all messages matching the
above criteria that were created with this instance as the
parent. The newest 'count' messages (that are newer than time)
are retrieved and then returned in order such that the first
message in the returned list is the oldest.

Note that the first message returned is not necessarily the
oldest one that is newer than time. This can occur if the
number of matching messages is greater than 'count' since the
'count' newest are selected before their order is reversed.

""
return [message.to_dictionary() for message in
    self.get_messages_query(message_type, recipient,
        time = time).fetch(count)[::-1]]

def get_messages_query(self, message_type, recipient,
                          time = datetime.min, sender = None,
                          keys_only = False):
    """Return a message query from this instance.

    Args:
        message_type: The message type to retrieve. If left as None or
        the empty string then all messages matching other criteria
        will be returned.
        recipient: The recipient of the messages to retrieve. All
        messages sent with a recipient of the empty string will also
        be retrieved.
        time: All messages retrieved must have been created after this
time.
        sender: The sender of the message.
        keys_only: If keys_only is set to true, this will only search
for messages that have recipient = recipient. Thus, it will
only include messages sent with no recipient if recipient
is set to ''. 

    Returns:
        A query object that can be fetched or further modified.
    ""
    query = Message.all(keys_only = keys_only)
    query.ancestor(self.key())
    query.filter('date >', time)
    if message_type is not None and message_type != '':
        query.filter('msg_type =', message_type)
    if sender:
        query.filter('sender =', sender)
    # Avoid doing two queries when we don't need to.
    if recipient == '':
        query.filter('recipient =', '')
    else:
        if keys_only:
            query.filter('recipient =', recipient)
        else:
def delete_messages(self, mtype = None):
    """Delete messages of a specified kind.
    """
    Args:
        type: A string of the message type to delete.
    Due to timeout issues with App Engine, this method will currently
    only succeed when running on App Engine if the number of messages
    being deleted is relatively small (~hundreds). It will attempt to
delete up to 1000. The timeout retry wrapper (see
game_server/autoretry_datastore.py) and using keys only search
drastically increases the chances of success, but this method is
still not guaranteed to complete.

    For more information see:
    http://groups.google.com/group/google-appengine/
browse_thread/thread/ee0800a3ca92fe69?pli=1
    http://stackoverflow.com/questions/108822/
delete-all-data-for-a-kind-in-google-app-engine
    """
    if mtype:
        db.delete(Message.all(keys_only = True).filter('msg_type =', mtype).
            ancestor(self.key()).order('date').fetch(1000))
        db.delete(Message.all(keys_only = True).ancestor(self.key()).order
            ('date')
            .fetch(1000))
    def check_player(self, pid):
        """Confirm that a player is currently in the instance.
        """
        Args:
            pid: A string containing the player's email address.
        Returns:
            The email address of the player.
        Raises:
            ValueError if the player is not in this instance.
        """
        player = utils.check_playerid(pid)
        if player in self.players:
            return player
        raise ValueError("%s is not in instance %s" % (pid, self.key().
            name()))
    def check_leader(self, pid):
        """Confirm that a player is the leader of the instance.
        """
        Args:
pid: A string containing the player's email address.

Returns:
The email address of the leader if pid contains it.

Raises:
ValueError if the player is not the leader of this instance.

player = utils.check_playerid(pid)
if player == self.leader:
    return player
raise ValueError("You must be the leader to perform this operation.")

def add_player(self, player):
    """Add a new player to this instance.
    
    Args:
        player: The email address of the player to add.
    
    A player can join a game instance if it is not full and either the
    instance is public or the player has been invited. If the player
    is already in the game instance this will succeed without
    modifying the instance.
    
    Raises:
        ValueError if the player is not already in the game and is
        unable to join.
    ""
    if player not in self.players:
        if player not in self.invited and not self.public:
            raise ValueError("%s not invited to instance %s." % (player, self.key().name()))
        if self.full:
            raise ValueError("%s could not join: instance %s is full" % (player, self.key().name()))
        if player in self.invited:
            self.invited.remove(player)
            self.players.append(player)
            self.setfull()
class Message(db.Expando):
    """A model for a message sent to a player in a game instance.
    Messages are used to pass information from player to player and from
    server to player. A Message's parent is the GameInstance which it is
    created for.
    Attributes:
    msg_type: A string that acts as a key for the message.
    recipient (optional): The intended recipient of this message.
    content: A JSON string that represents the contents of the message.
    date: The date of creation, automatically set upon instantiation.
    sender: A string describing the creator of the message.
    """
    msg_type = db.StringProperty(required=True)
    recipient = db.StringProperty(required=False)
    content = db.TextProperty(required=False)
    date = db.DateTimeField(required=True, auto_now_add=True)
    sender = db.StringProperty(required=True)

def to_dictionary(self):
    """Return a Python dictionary of the message.
    Returns a dictionary of the message:
    
    type: msg_type
    mrec: recipient
    contents: the Python representation of the content JSON string.
    mtime: The iso8601 string representation of the creation time of
    the message.
    msender: sender
    """
    return {'type': self.msg_type, 'mrec' : self.recipient,
    'contents' : simplejson.loads(self.content),
    'mtime' : self.date.isoformat(),
    'msender' : self.sender}

def to_json(self):
    """Return a json representation of the dictionary of this message
    """
    return simplejson.dumps(self.to_dictionary())
def get_content(self):
    """Return the Python representation of the contents of this
    message. """
    return simplejson.loads(self.content)

A.1.2 Extensions

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

# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"

authors = ['"Bill Magnuson" <billmag@mit.edu>']

import random
from django.utils import simplejson
from game-server.models.message import Message
from gameserver.utils import getboolean
from google.appengine.ext import db

def set_deck_command(instance, player, arguments):
    """Set the instance's deck to a new list of cards.
    """
    Args:
instance: The GameInstance database model for this operation.
player: The email address of the player requesting the action.
For this command, the player must be the current leader of the
instance.
arguments: A list of the cards to set the deck to.

Resets the deck used by card games from a standard 52 card deck to
the deck specified by the arguments list. A new deck can only be set
when no other card game methods have been invoked for a particular
game instance. The deck will remain the same throughout the life of
the game instance.

Returns:
The number of cards in the new deck.

Raises:
A ValueError if the requesting player is not the leader of the
instance.

""
instance.check_leader(player)
return set_deck(instance, arguments)

def deal_cards_command(instance, player, arguments):
    """Deal cards to players.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting the action.
        The player must be the current leader of the instance.
    arguments: A list of arguments to this command as explained below.

    The arguments list for this command consists of five items in order:
    1: cards_to_deal - The number of cards to deal as an integer.
    2: shuffle_deck - A boolean controlling whether or not the deck
        should be shuffled before the new hands are dealt. If the deck
        is shuffled, all hands are also cleared regardless of the value
        of is_new_hand.
    3: is_new_hand - A boolean indicating whether this is a new hand
        or not. If it is a new hand, then all hands will be cleared
        before the new cards are dealt. If the deck is shuffled before
        the cards are dealt then the hands are cleared automatically
        and this has no effect.
    4: ignore_empty_deck - Another boolean controlling whether to
        ignore an empty deck or not. If it is true, then cards will be
        dealt until the deck runs out and then this command will return
        successfully. If it is false, an error will occur if the deck
        runs out of cards.
    5: A list of player id's to be dealt to in the order to deal to
        them. Cards will be dealt one at a time to players in the order
        that they appear in this list.

    Cards are dealt to players using the instance's deck according to
    the arguments specified above. The cards are dealt in the order
determined by the last shuffling. Until a deck is re-shuffled, cards
will be dealt as if they were removed from the top of the deck and
given to the player permanently.

Returns:
The hand of the requesting player after cards are dealt.

Raises:
An IndexError if the deck runs out of cards and empty deck errors
are not being ignored.
A ValueError if any of the player id's in the list of players to
deal to are not in the game instance.
A ValueError if the requesting player is not the leader of the
instance.

```
instance.check_leader(player)
cards_to_deal = int(arguments[0])
shuffle = get_boolean(arguments[1])
if shuffle:
    shuffle_deck(instance)
is_new_hand = get_boolean(arguments[2])
ignore_empty_deck = get_boolean(arguments[3])
hands = deal_cards(instance, cards_to_deal, is_new_hand,
                   ignore_empty_deck,
                   arguments[4])
return hands[player]
```

def draw_cards_command(instance, player, arguments):
    """Draw cards from the deck and put them into the calling player's
    hand.
    
    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting the action.
    arguments: A list of arguments to this command as explained below.
    
The arguments list for this command consists of two items in order:
1: cards_to_draw - The number of cards to attempt to draw.
2: ignore_empty_deck - A boolean controlling whether to ignore an
   empty deck or not. If it is true, cards can be drawn until the
deck runs out and then this command will return
   successfully. If it is false, an error will occur if the deck
   runs out of cards and no changes will be made to the hand of
   the player.

    Returns:
    The hand of the player after drawing the new cards.
    
    Raises:
    An IndexError if the deck runs out of cards and empty deck errors
    are not being ignored.
    A ValueError if the requesting player is not in the instance.
    """
cards_to_draw = int(arguments[0])
ignore_empty_deck = get_boolean(arguments[1])
return draw_cards(instance, player, cards_to_draw, ignore_empty_deck)

def discard_command(instance, player, arguments):
    """ Remove the specified cards from the calling player's hand.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting the action.
    arguments: A list of cards to discard.

    Discarded cards are removed from a player's hand permanently. They
    are not re-added to the deck of cards to be dealt to other
    players. However, once a deck is shuffled, all cards become
    available again including any that have been discarded.

    If a player tries to discard a card that is not in their hand on the
    server, the request to discard that particular card is ignored, but
    the execution of the command continues.

    Returns:
    The current hand of the requesting player.
    """
    return discard(instance, player, arguments)

def pass_cards_to_player_command(instance, player, arguments):
    """ Remove cards from the calling player's hand and add them to
    another hand.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting the action.
    arguments: A list of two items. The first item is the email
    address of the player to pass the cards to and the second is a
    list of cards to pass to them.

    Raises:
    A ValueError if the player to pass the cards to is not in the game
    instance.

    """
    hands = get_hand_dictionary(instance)
to_player = instance.check_player(arguments[0])
return pass_cards(instance, player, to_player, arguments[1])

def get_cards_remaining_command(instance, player, arguments = None):
    """ Return the number of cards left in this deck to deal.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting the action.
    arguments: Not used, can be any value.

    Returns:
The number of cards that can still be dealt before the deck is empty. If the deck has not been set or no cards have been dealt (in the case that the default deck is being used), returns -1.

```python
return cards_left(instance)
```

------

# Helpers #

```python
def get_deck(instance):
    """Return the deck for this instance.
    
    Args:
    instance: The GameInstance database model for this operation.
    
    Returns:
    The current deck for this instance. If no deck exists, returns the default deck, unshuffled.
    ""
    if 'crd_deck_index' not in instance.dynamic_properties():
        instance.crd_deck_index = 0
    if 'crd_deck' not in instance.dynamic_properties():
        instance.crd_deck = [simplejson.dumps(card) for card in default_deck]
    return instance.crd_deck

def set_deck(instance, deck):
    """Set the deck for this instance to a new one.
    
    Args:
    instance: The GameInstance database model for this operation.
    deck: A list of cards to set as a new deck.
    
    Returns:
    The number of cards in the new deck.
    
    Raises:
    AttributeError if a deck has already been created for this instance.
    ""
    if 'crd_deck' in instance.dynamic_properties():
        raise AttributeError('Deck can only be set as the first operation in a card game.')
    instance.crd_deck = [simplejson.dumps(card) for card in deck]
    instance.crd_deck_index = 0
    return len(instance.crd_deck)

def get_hand_dictionary(instance):
    """Return a dictionary with the hands of each player in the instance.
    
    Args:
    ""
```
instance: The GameInstance database model for this operation.

Returns:
A dictionary with a list for each player in the game. Each player's list will include the cards currently in their hand. Keys in the dictionary are the email addresses of players.

if 'crd_hands' not in instance.dynamic_properties():
    return get_empty_hand_dictionary(instance)
else:
    return simplejson.loads(instance.crd_hands)

def get_empty_hand_dictionary(instance):
    """Return a dictionary with an empty hand for each player in the instance.
    ""
    hands = {}
    for player in instance.players:
        hands[player] = []
    return hands

def set_hand_dictionary(instance, hands, send_messages = True):
    """Set the hands of all players and send new hand messages.
    ""
    Args:
        instance: The GameInstance database model for this operation.
        hands: A dictionary containing the hand of each player in the game.
        send_messages: Whether or not to send a message to each player with their new hand.
    Stores the hands dictionary in the game instance. If send_messages is True, this will also send a new 'crd_hand' message to each player with their new hand.
    ""
    if send_messages:
        message_list = []
        for player in instance.players:
            message_list.append(instance.create_message(player, 'crd_hand', player, hands[player]))
        db.put(message_list)
    instance.crd_hands = db.Text(simplejson.dumps(hands))

def get_player_hand(instance, player):
    """Get the hand of a single player in an instance.
Args:
  instance: The GameInstance database model for this operation.
  player: The email address of the player.

Returns:
The list of cards that the player has or an empty list if they
do not have a hand.

Raises:
  ValueError if the player is not in the instance.

""
player = instance.check_player(player)
hands = get_hand_dictionary(instance)
return hands.get(player, [])

def set_player_hand(instance, player, hand, send_message = True):
  """ Set the hand of a single player in an instance.
  
  Args:
      instance: The GameInstance database model for this operation.
      player: The email address of the player.
      hand: The new hand of the player.
      send_message: Whether to send player a 'crd_hand' message
          with their new hand.
  
  Stores the new hands dictionary with the updated hand for player.
  If send_message is True, a message will be sent to player with
  their new hand.
  
  Raises:
      ValueError if the player is not in the instance.
  ""
  player = instance.check_player(player)
hands = get_hand_dictionary(instance)
hands[player] = hand
set_hand_dictionary(instance, hands, send_messages = False)
if send_message:
    instance.create_message(player, 'crd_hand', player, hand).put()

def get_next_card(instance):
  """ Return the card in the deck.
  
  Args:
      instance: The GameInstance database model for this operation.
  
  Returns:
      The Python representation of the next card in the deck. Because
      cards are stored as JSON strings they are first decoded before
      being returned.
  
  Raises:
      ValueError if the JSON decoding fails.
      IndexError if the deck has run out of cards.
  """
if cardsleft(instance) == 0:
    raise IndexError('Deck is empty')
card = simplejson.loads(get_deck(instance)[instance.crd_deck_index])
instance.crd_deck_index = instance.crd_deck_index + 1
return card

def shuffle_deck(instance):
    """Shuffle the deck and reset all hands."
    Args:
        instance: The GameInstance database model for this operation.
    Shuffles all cards in the original deck and makes them available to
    be dealt or drawn again. Also clears all players hands.
    Returns:
        The number of cards in the deck.
    """
    deck = get_deck(instance)
    random.shuffle(deck)
    instance.crd_deck_index = 0
    instance.crd_deck = deck
    set_hand_dictionary(instance, get_empty_hand_dictionary(instance))
    return len(instance.crd_deck)

def pass_cards(instance, from_player, to_player, cards):
    """Pass cards from one player to another."
    Args:
        instance: The GameInstance database model for this operation.
        from_player: Email address of the player who is passing the cards.
        to_player: Email address of the player who is receiving the cards.
        cards: A list of cards to pass.
    Searches the hand of from_player for each card in cards and if it
    is present, transfers it to_player's hand. If a card is not present,
    it is ignored.
    Returns:
        The hand of from_player after passing the cards.
    """
    hands = get_hand_dictionary(instance)
    from_player = instance.check_player(from_player)
    to_player = instance.check_player(to_player)
    for card in cards:
        try:
            hands[from_player].remove(card)
            hands[to_player].append(card)
        except ValueError:
            pass
    set_hand_dictionary(instance, hands)
return hands[from_player]

def discard(instance, player, cards, send_message = True):
    """ Remove the specified cards from player's hand."

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player to discard cards from.
    cards: The cards to be discarded.
    send_message: Whether to send player a 'crd_hand' message
        with their new hand.

    Discarded cards are removed from a player's hand permanently. They
    are not re-added to the deck of cards to be dealt to other
    players. However, once a deck is shuffled, all cards become
    available again including any that have been discarded.

    If a player tries to discard a card that is not in their hand on the
    server, the request to discard that particular card is ignored, but
    the execution of the command continues.

    Returns:
    The hand of player after discarding the cards.

    Raises:
    ValueError if the player is not in the instance.
    """
        hand = get_player_hand(instance, player)
    for card in cards:
        try:
            hand.remove(card)
        except ValueError:
            pass
    set_player_hand(instance, player, hand, send_message)
    return hand

def deal_cards(instance, cards_to_deal, is_new_hand, ignore_empty_deck, deal_to):
    """ Deal cards to players."

    Args:
    instance: The GameInstance database model for this operation.
    cards_to_deal: The number of cards to deal as an integer.
    is_new_hand: A boolean indicating whether this is a new hand or
        not. If it is a new hand, then all hands will be cleared before
        the new cards are dealt. If the deck is shuffled before the
        cards are dealt then the hands are cleared automatically and
        this has no effect.
    ignore_empty_deck: Another boolean controlling whether to ignore
        an empty deck or not. If it is true, then cards will be dealt
        until the deck runs out and then this command will return
        successfully. If it is false, an error will occur if the deck
runs out of cards.

deal_to: A list of player id's to be dealt to in the order to deal
to them. Cards will be dealt one at a time to players in the
order that they appear in this list.

The cards are dealt in the order determined by the last
shuffling. Until a deck is re-shuffled, cards will be dealt as if
they were removed from the top of the deck and given to the player
permanently.

Returns:
The hand of the requesting player after cards are dealt.

Raises:
ValueError if a player in deal_to is not in the instance.
IndexError if the deck runs out of cards and ignore_empty_deck is
not True.

""
hands = {}
if not is_new_hand:
    hands = get_hand_dictionary(instance)

if cards_to Deal:
    deal_to = [instance.check_player(pid) for pid in deal_to]
    for player in deal_to:
        hands.setdefault(player, [])
    try:
        for i in xrange(cards_to deal):
            for player in deal_to:
                hands[player].append(get_next_card(instance))
        except IndexError:
            if not ignore_empty_deck:
                raise

    set_hand_dictionary(instance, hands)
return hands

def draw_cards(instance, player, cards_to_draw,
               ignore_empty_deck = True, send_message = True):
    """Draw cards from the deck and put them into player's hand.

    Args:
        instance: The GameInstance database model for this operation.
        player: The email address of the player to give the cards to.
        cards_to_draw: The number of cards to draw from the deck.
        ignore_empty_deck - A boolean controlling whether to ignore an
        empty deck or not. If it is true, cards can be drawn until the
deck runs out and then this command will return
        successfully. If it is false, an error will occur if the deck
        runs out of cards and no changes will be made to the hand of
        the player.
        send_message: Whether to send player a 'crd_hand' message
        with their new hand.
Returns:
The hand of the requesting player after they have drawn their cards.

Raises:
ValueError if player is not in the game instance.
IndexError if the deck runs out of cards and ignore_empty_deck is not True.

""
hand = get_player_hand(instance, player)
try:
    for i in xrange(cards_to_draw):
        hand.append(get_next_card(instance))
except IndexError:
    if not ignore_empty_deck:
        raise
set_player_hand(instance, player, hand, send_message)
return hand

def cards_left(instance):
    """ Return the number of cards left to deal before a shuffle is required.
    Args:
    instance: The GameInstance database model for this operation.
    """
    if 'crd_deck' not in instance.dynamic_properties():
        return -1
    return len(instance.crd_deck) - instance.crd_deck_index
All scores are stored as integers with the convention that higher numbers are better.


__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

import operator
from django.utils import simplejson
from google.appengine.ext import db

###########
# Server command functions #
###########

def get_scoreboard_command(instance, player, arguments = None):
    """ Get the current scoreboard.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting this action.
    arguments: Not used, can be any value.

    Returns:
    The complete scoreboard as a list of [score, email] lists for each player in the game. The scoreboard is sorted with the highest score first.
    """
    return format_scoreboard_for_app_inventor(get_scoreboard(instance))

def get_score_command(instance, player, arguments):
    """ Set the score of a single player.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting this action.
    arguments: A one item list containing the player id of the player to get the score of.

    Returns:
    The score of the requested player.

    Raises:
    ValueError if the player in arguments is not in the game.
    """
    get_score_player = instance.check_player(arguments[0])
    return get_score(instance, get_score_player)

def set_score_command(instance, player, arguments):
    """ Set a player's score to a new value.

    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player to set the score of.
    arguments: A list of two items. The first item is the player id
of the player who's score is to be set. The second item is the
integer to set that player's score to.

Returns:
The complete scoreboard after setting the new score value.

Raises:
ValueError if the specified player is not in the instance.

player = instance.check_player(arguments[0])
new_score = int(arguments[1])
board = set_score(instance, player, new_score)
return format_scoreboard_for_app_inventor(board)

def add_to_score_command(instance, player, arguments):
    """ Change a player's score by an integer amount.
    ""
    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player to add points to.
    arguments: A list of two items. The first item is the player id
    of the player who's score is to be set. The second item is the
    integer amount to change that player's score by. This value can
    be positive or negative.
    
    In order for this operation to work correctly scores must be
    represented in the scoreboard as single integer items.
    
    Returns:
    The complete scoreboard after adding to player's score.
    
    Raises:
    ValueError if the specified player is not in the instance.
    ValueError if the specified score cannot parse correctly.
    ""
    player = instance.check_player(arguments[0])
    delta = int(arguments[1])
    board = add_to_score(instance, player, delta)
    return format_scoreboard_for_app_inventor(board)

def clear_scoreboard_command(instance, player, arguments = None):
    """ Reset all scores to 0.
    ""
    Args:
    instance: The GameInstance database model for this operation.
    player: The email address of the player requesting this action.
    arguments: Not used, can be any value.
    
    Returns:
    An empty scoreboard with a score of 0 for each player.
Raises:

ValueError if player is not the leader of this instance.

```
instance.check_leader(player)
instance.scoreboard = '{}'
return get_scoreboard_command(instance, player, arguments)
```

###########
# Helpers #
###########

def get_score(instance, player):
    """Get a player’s score.
    Args:
        instance: The instance to get the scoreboard from.
        player: The player to check the score of.
    Returns:
        The player’s score.
    Raises:
        ValueError if the player is not in the instance.
    ""
    player = instance.check_player(player)
    board = get_scoreboard(instance)
    return board[player]

def set_score(instance, player, new_score):
    """Set a player’s score.
    Args:
        instance: The game instance to modify the scoreboard of.
        player: The player to set the score of.
        new_score: An integer to set their score to.
    Returns:
        The scoreboard as a dictionary after setting a new value for
        player’s score.
    Raises:
        ValueError if the player is not in the instance.
    ""
    player = instance.check_player(player)
    scoreboard = get_scoreboard(instance)
    scoreboard[player] = new_score
    instance.scoreboard = simplejson.dumps(scoreboard)
    return scoreboard

def add_to_score(instance, player, delta):
    """Change a player’s score by delta.
    Args:
        instance: The game instance to modify the scoreboard of.
        player: The player to set the score of.
        new_score: An integer to set their score to.
    Returns:
        The scoreboard as a dictionary after setting a new value for
        player’s score.
    Raises:
        ValueError if the player is not in the instance.
    ""
    player = instance.check_player(player)
    scoreboard = get_scoreboard(instance)
    scoreboard[player] = new_score
    instance.scoreboard = simplejson.dumps(scoreboard)
    return scoreboard
instance: The game instance to modify the scoreboard of.
player: The player to change the score of.
delta: The integer amount to change player's score by (can be negative).

In order for this operation to work correctly scores must be represented in the scoreboard as single integer items.

Returns:
The scoreboard as a dictionary after modifying player's score.

```python
player = instance.check_player(player)
scoreboard = get_scoreboard(instance)
if player in scoreboard:
    scoreboard[player] += delta
else:
    scoreboard[player] = delta
instance.scoreboard = db.Text(simplejson.dumps(scoreboard))
return scoreboard
```

def get_scoreboard(instance):
    """Get a dictionary of the scoreboard for the specified instance.
    
    Args:
    instance: The instance to get the scoreboard from.
    
    Returns:
    A dictionary with a score entry for each player in the instance. If no score was previously present, a value of 0 is entered.
    ""
    board = None
    if 'scoreboard' not in instance.dynamic_properties():
        board = {}
    else:
        board = simplejson.loads(instance.scoreboard)
        for player in instance.players:
            if not board.has_key(player):
                board[player] = 0
        return board

def format_scoreboard_for_app_inventor(board):
    """Return a scoreboard suitable to return to App Inventor.
    
    Args:
    board: The dictionary of scores for all players in the game.
    
    Returns:
    A list of [score, player email] lists ordered by highest score.
    """
    board_list = [[v,k] for k, v in board.items()]
    board_list.sort(key = operator.itemgetter(0), reverse = True)
    return board_list
A.2 Custom Modules

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

```python
# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
""
Defines the commands available from custom modules. Custom modules
differ from built in server commands or server extensions because they
are more narrowly focused on a particular game's functionality.

Custom modules will generally be built on a per game basis and
included when game creators deploy their own App Engine servers.

This file currently includes commands for custom modules meant to be
used as examples. These can be removed to decrease load time if they
are not being used.

""

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

from custommodules.androidstoandroids import ata_commands
from custommodules.bullsandcows import bac_commands
from custommodules.amazon import amazoncommands
from custommodules.voting import voting_commands

custom_command_dict = {
    # Androids to Androids
    'ata_new_game': ata_commands.new_game_command,
    'ata_submit_card': ata_commands.submit_card_command,
    'ata_end_turn': ata_commands.end_turn_command,

    # Bulls and Cows
    'bac_new_game': bac_commands.new_game_command,
    'bac_guess': bac_commands.guess_command,

    # Amazon
    'amz_keyword_search': amazoncommands.keyword_search_command,
    'amz_isbn_search': amazoncommands.isbn_search_command,

    # Voting
    'vot_cast_vote': voting_commands.cast_vote_command,
}```
A.2.1 Amazon

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

```
# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

""" Looks up books on Amazon by keyword or ISBN

Uses a library originally downloaded from
http://blog.umlglu.co.uk/blog/2009/jul/12/pyaws-adding-request-
authentication/
to access AWS E-Commerce Service API's and retrieve book results
for searches by keyword and ISBN number.
""

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>',
               '"Dave Wolber" <wolber@usfca.edu>']

# This file has had its license and secret keys removed and will not
# function.
license_key = ''
secret_key = ''

from pyaws import ecs

return_limit = 5

def keyword_search_command(model, player, arguments):
    """ Return books by keyword.
    Args:
        model: Not used, can be anything.
```
player: Not used, can be anything.
arguments: A one item list containing the keywords to search for.

Returns:
A list of three item lists. Each sublist represents
a result and includes the book title, its lowest found
price and its ASIN number.

""
return amazon_by_keyword(arguments[0])

def isbn_search_command(model, player, arguments):
    Args:
        model: Not used, can be anything.
        player: Not used, can be anything.
        arguments: A one item list containing the keywords to search for.
    Returns:
        A list with a single sublist representing the book found.
        The sublist contains the book title, its lowest found
        price and its ASIN number.
    Raises:
        ValueError if the ISBN number is invalid.
    """
return amazon_by_isbn(arguments[0])

def amazon_by_keyword(keyword):
    """ Use the ecs library to search for books by keyword.
    Args:
        keyword: A string of keyword(s) to search for.
    Returns:
        A list of three item lists. Each sublist represents
        a result and includes the book title, its lowest found
        price and its ASIN number.
    """
ecs.setLicenseKey(license_key)
ecs.setSecretKey(secret_key)
ecs.setLocale('us')
books = ecs.ItemSearch(keyword, SearchIndex='Books', ResponseGroup='Medium')
return format_output(books)

def amazon_by_isbn(isbn):
    """ Use the ecs library to search for books by ISBN number.
    Args:
        isbn: The 10 digit ISBN number to look up.
    Returns:
A list with a single sublist representing the book found. The sublist contains the book title, its lowest found price and its ASIN number.

Raises:
ValueError if the ISBN number is invalid.

ecs.setLicenseKey(license_key)
ecs.setSecretKey(secret_key)
ecs.setLocale('us')

try:
    books = ecs.ItemLookup(isbn, IdType='ISBN', SearchIndex='Books',
                           ResponseGroup='Medium')
    return format_output(books)
except ecs.InvalidParameterValue:
    raise ValueError('Invalid ISBN')

def format_output(books):
    """ Return a formatted output list from an iterator returned by ecs. ""

    Args:
    books: An iterator of book results from the ecs library.

    Returns:
    A list of three item lists. Each sublist represents a result and includes the book title, its lowest found price and its ASIN number.

    
    size = min(len(books), return_limit)
    return [[books[i].Title, get_amount(books[i]), books[i].ASIN]
            for i in xrange(size)]

def get_amount(book):
    """ Return the lowest price found or 'Not found.' if none exists. ""

    try:
    except:
        return 'Not found.'

---

A.2.2 Androids to Androids

---

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

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

164
A set of server commands to implement Androids to Androids.

Androids to Androids is a card game played by at least three players. The first leader of the game is the creator of the game instance. The game proceeds in rounds with the winner of each round becoming the leader of the next round. To start the game, each player is dealt seven cards with nouns on them. These cards comprise their hand.

At the beginning of each round, an adjective or characteristic is chosen at random and sent to each player. Every player except the leader will then choose a card from their hand to submit for the round. Upon submission their hand will be replenished with another card so that they always have seven cards in their hand.

The leader will then choose a single noun card from those submitted by the other players in response to the characteristic. The leader can use any criteria they wish to select the card that should win the round, however, they are not allowed to know the identity of the person that submits each card.

Once a winner is chosen, a new round is started with the previous winner as the new leader. Play continues in this way until one of the players reaches a predetermined winning score and is declared the winner.

Each command returns information that is immediately useful to the player who requested the command. In addition, any changes to their hand or the set of cards players have submitted will be sent to them via message so that they can easily recover state if they lose their active session in the game.

Submitting cards and ending turns both require that the player submit the round number that they intend for that action to apply to. If that number does not match the current round the action will be ignored and the command will return information to allow that player to get back up to date with the game.

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

import random
import decks
from game_server.extensions import scoreboard
from game_server.extensions import card_game
from google.appengine.ext import db
from django.utils import simplejson
hand_size = 7
winning_score = 5
min_players = 3

###############
# Game Commands #
###############

def new_game_command(instance, player, arguments = None):
    """ Start a new game of Androids to Androids.

    Args:
        instance: The GameInstance database model for this operation.
        player: The player starting the game. Must be the current leader
        of the instance instance.
        arguments: Not used, can be any value.

    Closes the game to new players, deals a new hand to each player and
    selects a new characteristic card to begin round 1. Sends a new
game message to all players with the starting card and the empty
scoreboard.

    Each player will also receive a message of type crd_hand that
    contains all of the cards dealt to them.

    Returns:
        A three item list consisting of the new characteristic card for
        this turn, the current (empty) scoreboard, and the requesting
        player's current hand.

    Raises:
        ValueError if an Androids to Androids game is already in
        progress, if player is not the current leader of the game or if
        there are not enough players in the game to begin.
    """
    player = instance.check_leader(player)
    instance.public = False
    if 'ata_round' in instance.dynamic_properties():
        raise ValueError("This game is already in progress. " +
                        "Please refresh the game state.")
    if len(instance.players) < min_players:
        raise ValueError("Androids to Androids requires at least %d
                        players." % min_players)
    instance.max_players = len(instance.players)
    try:
        card_game.set_deck(instance, decks.nouncards)
    except AttributeError:
        pass
    card_game.shuffle_deck(instance)
hands = card_game.deal_cards(instance, hand_size, True, False, instance.players)

instance.starting_players = instance.players
instance.ata_round = 0
setup_new_round(instance)
board = scoreboard.clear_scoreboard_command(instance, player)
instance.create_message(instance.leader, 'ata_new_game', '',
                        [instance.ata_char_card, board]).put()
return [instance.ata_char_card, board, hands[player]]

def submit_card_command(instance, player, arguments):
    """ Submit a noun card for the current round.

    Args:
    instance: The GameInstance database model for this operation.
    player: The player submitting the card. Cannot be the leader.
    arguments: A two item list consisting of the round to submit this
card for and the card itself.

    If the submission is for the wrong round, a four item list with an
error string as its first element will be returned. The remaining
elements are the player's hand, the current round and the current
characteristic card to respond to. No other action will be taken.

    Removes the indicated card from the player's hand and adds it
to this round's submissions. The current submissions are sent via
message to all players.

    The requesting player's hand will be dealt another card after
removing the submitted one. The updated hand will be sent to the
requesting player in a message and be included in the return value
of this command.

    Returns:
    If the submission is for the correct round, returns a three item
list consisting of the current round number, a list of the
submissions made so far by other players in this round and the
player's new hand.

    Raises:
    ValueError if player is the leader. The leader is not allowed to
submit cards.
"""

    if int(arguments[0]) != instance.ata_round:
        hand = card_game.get_player_hand(instance, player)
        return ['You tried to submit a card for the wrong round. ' +
                'Please try again.', hand, instance.ata_round,
                instance.ata_char_card]
    missing_player = check_players(instance)
    if missing_player:
        return missing_player

167
if player == instance.leader:
    raise ValueError("The leader may not submit a card.")

submission = arguments[1]
submissions = set_submission(instance, player, submission).values()
instance.create_message(player, 'ata_submissions', '',
    [instance.ata_round, submissions, submission]).put()

card_game.discard(instance, player, [submission], False)
hand = card_game.draw_cards(instance, player, 1)
return [instance.ata_round, submissions, hand]

def end_turn_command(instance, player, arguments):
    """End the current turn and start a new one.

    Args:
        instance: The GameInstance database model for this operation.
        player: The player submitting the card. Must be the current
            leader.
        arguments: A two item list consisting of the round number to end
            and the selected winning card.

    If the command is for the wrong round, a four item list with an
    error string as its first element will be returned. The remaining
    elements are the player's hand, the current round and the current
    characteristic card to respond to. No other action will be taken.

    Ends the current turn and adds 1 point to the score of the player
    who submitted the winning card. If that player has reached the
    winning score, an 'ata_game_over' message will be sent to all
    players. The game over message content will be a three item list as
    its contents. The list contains the final round number, the winning
    card and the final scoreboard.

    Otherwise, sends an 'ata_new_round' message to all players. The new
    round message contents will be a five item list with the round
    number, the new characteristic card, the previous round winner, the
    winning card and the current scoreboard.

    Returns:
        If the command was for the correct round, returns the content of
        whichever message was sent to all players as described above.
        Raises:
        ValueError if player is not the leader.
        KeyError if no player has submitted the winning card.
    """
    if int(arguments[0]) != instance.ata_round:
        hand = card_game.get_player_hand(instance, player)
        return ['You tried to end a turn that has already ended. ' +
            'Please try again.', hand, instance.ata_round,
            instance.ata_char_card]
missing_player = check_players(instance)
if missing_player:
    return missing_player
instance.check_leader(player)
card = arguments[1]
winner = None
for player, submitted_card in get_submissions_dict(instance).items():
    if card == submitted_card:
        winner = player
        break
if winner == None:
    raise KeyError('No player has submitted the card %s.' % card)
board = scoreboard.add_to_score(instance, winner, 1)

# Check to see if anyone has won
instance.leader = winner
if board[winner] == winning_score:
    return endgame(instance, card)
setup_new_round(instance)
return_scoreboard = scoreboard.format_scoreboard_for_app_inventor(board)
content = [instance.ata_char_card, return_scoreboard,
           instance.ata_round, winner, card]
instance.create_message(instance.leader, 'ata_new_round', '',
content).put()
return content

###########
# Helpers #
###########
def check_players(instance):
    """ Checks to see if any of the starting players have left.
    Args:
        instance: The GameInstance model for this operation.
    If a player has left the game, they are invited back and
    a ValueError is raised.
    Raises:
        ValueError if a player has left the game.
    """
    if len(instance.players) < len(instance.starting_players):
        for starting_player in instance.starting_players:
            if starting_player not in instance.players:
                instance.invited.append(starting_player)
        return ('%s left during your game. They have %s left during your game. They have been invited and must rejoin before continuing.')
    return False
def end_game(instance, winning_card):
    """End the current game and inform all players of the winner.
    
    Args:
    instance: The GameInstance database model for this operation.
    winning_card: The card chosen as the winner of the final round.
    
    Sends an 'ata_game_over' message from the winner to all players with
    a three item list as its contents. The list contains the final round
    number, the winning card and the final scoreboard.
    
    Deletes the ataround, ata_char_card and ata_submissions properties
    from the GameInstance database model to allow for a new game to be
    player in this same instance with the previous winner as the new
    leader.
    
    Returns:
    The content of the message sent to all players.
    """
    content = [instance.ataround, winning_card, 
               scoreboard.get_scoreboard_command(instance, instance. 
               leader)]
    instance.create_message(instance.leader, 'atagameover', ' 
    content).put()
    del instance.ataround 
    del instance.ata_char_card 
    del instance.ata_submissions 
    instance.scoreboard = '{}' 
    return content 

def setup_new_round(instance):
    """Update the round number, char card and submissions for a new
    round.
    
    Args:
    instance: The GameInstance database model for this operation.
    
    Increments ataround, clears the submissions dictionary and sets
    ata_char_card to a new value from the list of characteristic cards.
    """
    instance.ataround += 1 
    instance.atasubmissions = db.Text('{}')

    new_card = random.choice(decks.characteristic_cards)
    if 'ata_char_card' in instance.dynamic_properties():
        while instance.ata_char_card == new_card:
            new_card = random.choice(decks.characteristic_cards)
            instance.ata_char_card = new_card 

def get_submissions_dict(instance):
    """Return a Python dictionary that maps cards to players.
    
    Args:
instance: The GameInstance database model for this operation.

Returns:
A Python dictionary of cards to players for all cards
submitted so far during this round.

""
return simplejson.loads(instance.ata_submissions)

def set_submission(instance, player, card):
""
Records the submission of card as coming from player.

Args:
instance: The GameInstance database model for this operation.
player: The player submitting the card.
card: The card to submit.

Returns:
A Python dictionary of cards to players for all cards
submitted so far during this round.

""
submissions = get_submissions_dict(instance)
if player in submissions:
    raise ValueError('You have already submitted a card for this round .')
submissions[player] = card
instance.ata_submissions = simplejson.dumps(submissions)
return submissions

A.2.3 Bulls and Cows

A version of bulls and cows using colors.

At the beginning of a new game a solution sequence of four colors is
randomly chosen from the set of colors. Each color appears at most
once in the solution. The player then makes guesses on the sequence
of colors in the solution. After each guess, they are informed of how
many 'cows' and 'bulls' they have in their guess. A 'bull' is when a player has the correct color in the correct position in their guess. A 'cow' is when a player has one of the correct colors, but it is in the wrong position.

Although the solution only includes each color once, a player is allowed to use the same color more than once in their guess. While obviously not correct, doing so might give the player information that they want about the solution.

The player begins with a score such that they will end with a score of zero if they guess completely wrong every time. After each guess is made, two points are deducted for each completely wrong color and one point is deducted for a correct color 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 are not awarded a score.

def new_game_command(instance, player, arguments = None):
    """Start a new game and reset any game in progress.

    Args:
    instance: The GameInstance database model for this operation.
    player: The player starting a new game. Must be the only player in the instance.
    arguments: Not used, can be any value.

    Returns:
    A list containing the number of guesses remaining, the starting score of the player, the player's historical high score and the number of games completed in the past.

    Raises:
    ValueError if there is more than 1 player in the instance or the player is not the current leader.
    """
old_games = instance.get_messages_query('bac_game', player,
                                          sender = player,
                                          keys_only = True)
db.delete(old_games)
score = scoreboard.get_score(instance, player)
if (score == 0):
    # Score is [high score, total score, games played]
    score = [0, 0, 0]
    scoreboard.set_score(instance, player, score)

    game = Message(parent = instance, sender = player,
                         msg_type = 'bac_game', recipient = player)
    game.bac_solution = sample(colors, solution_size)
    game.bac_guesses_remaining = starting_guesses
    game.bac_score = solution_size * starting_guesses * 2
    game.bac_last_guess = ['']
    game.bac_last_reply = ''
    game.put()
    return [game.bac_guesses_remaining, game.bac_score, score,
            game.key().id()]

def guess_command(instance, player, arguments):
    """ Evaluate a guess and determine the score."

    Args:
        instance: The GameInstance database model for this operation.
        player: The player making the guess. Must be the leader of
               the instance.
        arguments: A two element list containing the game id and a second
                  list with the guessed colors.

    Returns:
        If the player has guessed correctly:
        A two element list containing a score list and a boolean of
        whether or not this game set a new high score. The score list is
        a three element list containing the player's high score, their
        total score and their total number of games played.

        Otherwise:
        A four element list containing the player's remaining score, the
        number of guesses remaining, the number of bulls for this guess
        and the number of cows for this guess.

    Raises:
        ValueError if the player is not the current instance leader and
        only member of the game.
        ValueError if the player has no guesses remaining.
        ValueError if the guess does not have the correct number of
        elements.
        ValueError if no game has been started yet.

    """
    guess = arguments[1]
    if len(guess) != solution_size:
        raise ValueError("Guess was not the right number of elements.")
game = db.get(Key.from_path('Message', int(arguments[0])),
          parent = instance.key())

if game is None:
  raise ValueError("Game not found. Please start a new game."
if game.sender != player:
  raise ValueError("This is not your game. Please start a new game.")

if guess == game.bac_last_guess:
  return simplejson.loads(game.bac_last_reply)

if game.bac_guesses_remaining == 0:
  raise ValueError("No turns left, please start a new game.")

return_content = None

if guess == game.bac_solution:
  game.bac_guesses_remaining = 0
  new_high_score = False
  score = scoreboard.get_score(instance, player)
  if game.bac_score > score[0]:
    new_high_score = True
    score[0] = game.bac_score
  score[1] = score[1] + game.bac_score
  scoreboard.set_score(instance, player, score)
  return_content = [score, new_high_score]
else:
  game.bac_guesses_remaining -= 1
  bulls = cows = 0
  for i in xrange(solution_size):
    if guess[i] == game.bac_solution[i]:
      bulls += 1
    elif guess[i] in game.bac_solution:
      cows += 1

  score_deduction = solution_size * 2 - cows - 2 * bulls
  game.bac_score -= score_deduction
  return_content = [game.bac_guesses_remaining, game.bac_score,
                   bulls, cows]
  game.bac_last_reply = simplejson.dumps(return_content)
  game.bac_last_guess = guess
  game.put()
return return_content

A.2.4 Voting


# Copyright 2010 Google Inc.
# Licensed under the Apache License, Version 2.0 (the "License");
The commands are split into two categories. The first is for people who are performing the voting. The second category is for the creation and management of polls.

Voting:
Players find out about new polls by retrieving messages with types 'poll' or 'closed_poll' from the instance.

Once a player has found out about polls, they can cast votes and get results for closed polls and polls they have already voted in.

When a player votes in a poll they immediately receive the current results of that poll. They will be able to fetch those results until the poll creator deletes the poll.

Poll Management:
The remaining commands are for managing polls. Polls can be created, closed and deleted. Players can get the polls they have created with the get my polls command.

__authors__ = ['"Bill Magnuson" <billmag@mit.edu>']

from django.utils import simplejson
from game_server.models.message import Message
from google.appengine.ext import db

def castvotecommand(instance, player, arguments):
    """Cast a vote in a poll and return its current results.

    Args:
    instance: The parent GameInstance model of this poll.
    player: The player that is casting a vote.
    arguments: A two item list of the poll id and the zero based index of the option to select.

    Returns:
    A two item list containing a message and the current votes for the poll. The message will be one of:
    Your vote was already counted in this poll.
    Poll closed to new votes.
Vote accepted.

Raises:
ValueError if the vote index is larger than the number of options.
ValueError if the player is not in the instance.

instance.check_player(player)
poll = get_poll(instance, arguments[0])

if not poll.open:
    return ['Poll closed to new votes.', poll.votes]
if player in poll.voters:
    return ['Your vote was already counted in this poll.', poll.votes]

try:
    poll.voters.append(player)
    vote_index = int(arguments[1])
    poll.votes[vote_index] += 1
    poll.put()
except ValueError:
    raise ValueError('Invalid vote choice.')
return ['Vote accepted.', poll.votes]

def get_results_command(instance, player, arguments):
    """Gets the results of a poll.

    Args:
    instance: The parent GameInstance model of the poll.
    player: The player requesting the results.
    arguments: A one item list containing the id number of the poll.

    Returns:
    If the player has not voted in this poll and it is still open,
    this will return a single item list with a message for the
    requesting player.
    Otherwise returns a list with information about the poll. See
    get_poll_return_list for its format.

    Raises:
    ValueError if the player is not in the instance.
    ""
instance.check_player(player)
poll = get_poll(instance, arguments[0])
if not poll.open:
    return ['Poll is now closed.', poll.votes]
if player in poll.voters:
    return ['You have already voted in this poll.', poll.votes]
return ['You have not voted in this poll yet.]

def make_new_poll_command(instance, player, arguments):
    """Make a new poll.
Args:

instance: The game instance to add the poll to.
player: The email of the player creating the poll.
arguments: A two item list containing the question and a second list of 2–5 options.

Returns:

Returns a list with information about the poll just created.
See get_poll_return_list for its format.

Raises:

ValueError if the player is not in the instance.

""
instance.check_player(player)
if not arguments[0]:
    raise ValueError('Question cannot be empty')
size = len(arguments[1])
if size < 2 or size > 5:
    raise ValueError('Incorrect number of options for poll. ' +
        'Must be between two and five.')
poll = Message(parent = instance, sender = player,
    msg_type = 'poll', recipient = '')
poll.put()
arguments.append(poll.key().id())
poll.content = simplejson.dumps(arguments)
poll.votes = [0] * size
poll.open = True
poll.voters = ['',
poll.put()
return get_poll_return_list(poll)

def close_poll_command(instance, player, arguments):
    """Close an existing poll.
   "
    Args:
    instance: The parent GameInstance model of the poll.
    player: The email of the player closing the poll. Must be the poll's creator.
    arguments: A one argument list with the poll's id number.

    Returns:
    A list with information about the poll just closed. See get_poll_return_list for its format.

    Raises:
    ValueError if player is not the creator of the poll.
    ValueError if the player is not in the instance.
    ""
    instance.check_player(player)
poll = get_poll(instance, arguments[0])
if poll.sender != player:
    raise ValueError('Only the person that created this poll may close
poll.open = False
poll.msg_type = 'closed_poll'
poll.put()
return get_poll_return_list(poll)

def delete_poll_command(instance, player, arguments):
    """Delete an existing poll.
    ""
    Args:
    instance: The parent GameInstance model of the poll.
    player: The email of the player closing the poll. Must be the
           poll's creator.
    arguments: A one argument list with the poll's id number.
    Returns:
    True if the deletion is successful.
    Raises:
    ValueError if player is not the creator of the poll.
    ValueError if the player is not in the instance.
    """
    instance.check_player(player)
poll = get_poll(instance, arguments[0])
if poll.sender != player:
    raise ValueError('Only the person that created this poll may
                     delete it.')</ndb.delete(poll)
return [True]

def get_poll_info_command(instance, player, arguments):
    """Get information about an existing poll.
    ""
    Args:
    instance: The parent GameInstance model of the poll.
    player: The email of the player requesting information. Must
            be the poll's creator.
    arguments: A one argument list with the poll's id number.
    Returns:
    A list with information about the poll. See
    get_poll_return_list for its format.
    Raises:
    ValueError if player is not the creator of the poll.
    Raises:
    ValueError if the player is not in the instance.
    """
    instance.check_player(player)
poll = get_poll(instance, arguments[0])
if poll.sender != player:
    raise ValueError('Only the person that created the poll can'
                     + 'request its information.')
return get_poll_return_list(poll)
def get_my_polls_command(instance, player, arguments = None):
    
    """ Get the polls created by a player in the instance.
    
    Args:
        instance: The parent GameInstance model of the polls.
        player: The email of the player requesting the polls.
        arguments: Not used, can be any value.
    
    Finds all polls created by this player.
    
    Returns:
        A list of two item lists with each containing the
        id number of the poll and its question.
    
    Raises:
        ValueError if the player is not in the instance.
    """
    instance.check_player(player)
    query = instance.get_messages_query('', '', sender = player)
    polls = query.fetch(1000)
    return [[poll.key().id(), poll.get_content()[0]] for poll in polls[:-1]]

def get_poll(instance, argument):
    """ Get a poll database model.
    
    Args:
        instance: The parent GameInstance database model of the poll.
        argument: The poll id argument from the server command
            arguments list.
    
    Returns:
        A Message database model of the poll.
    
    Raises:
        ValueError if argument fails to parse to an int or the
        poll doesn't exist in the database.
    """
    try:
        poll_id = int(argument)
    except ValueError:
        raise ValueError('Poll id failed to parse to a number.')
    poll_key = db.Key.from_path('Message', poll_id,
                                parent = instance.key())
    poll = db.get(poll_key)
    if poll is None:
        raise ValueError('Poll no longer exists.')
    return poll

def get_poll_return_list(poll):
    """ Get a list to return to the GameClient component for a poll.
Args:

poll: A Message database model that is a poll.

Returns:

A list with the following five items:

The poll question.
The poll options as a list.
The poll id number.
The poll votes as a list.
Whether the poll is open.

```
content = poll.get_content()
content.extend([poll.votes, poll.open])
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 Game Client Component

```
// Copyright 2009 Google Inc. All Rights Reserved.

package com.google.devtools.simple.runtime.components.android;

import com.google.devtools.simple.common.ComponentCategory;
import com.google.devtools.simple.runtime.annotations.DesignerComponent;
import com.google.devtools.simple.runtime.annotations.DesignerProperty;
import com.google.devtools.simple.runtime.annotations.SimpleEvent;
import com.google.devtools.simple.runtime.annotations.SimpleFunction;
import com.google.devtools.simple.runtime.annotations.SimpleObject;
import com.google.devtools.simple.runtime.annotations.SimpleProperty;
import com.google.devtools.simple.runtime.annotations.UsesPermissions;
import com.google.devtools.simple.runtime.components.android.util.AsyncCallbackPair;
import com.google.devtools.simple.runtime.components.android.util.AsynchUtil;
import com.google.devtools.simple.runtime.components.android.util.GameInstance;
```
import com.google.devtools.simple.runtime.components.android.util.
        LoginServiceUtil;
import com.google.devtools.simple.runtime.components.android.util.
        PlayerListDelta;
import com.google.devtools.simple.runtime.components.android.util.
        WebServiceUtil;
import com.google.devtools.simple.runtime.components.util.JsonUtil;
import com.google.devtools.simple.runtime.components.util.YailList;
import com.google.devtools.simple.runtime.errors.YailRuntimeError;
import com.google.devtools.simple.runtime.events.EventDispatcher;
import android.app.Activity;
import android.os.Handler;
import android.util.Log;
import org.apache.http.NameValuePair;
import org.apache.http.message.BasicNameValuePair;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import java.util.ArrayList;
import java.util.List;

/**
 * GameClient provides a way for AppInventor applications to
 * communicate with online game servers. This allows users to create
 * games that are coordinated and managed in the cloud.
 * Most communication is done by sending keyed messages back and
 * forth between the client and the server in the form of YailLists.
 * The server and game client can then switch on the keys and perform
 * more complex operations on the data. In addition, game servers can
 * implement a library of server commands that can perform complex
 * functions on the server and send back responses that are converted
 * into YailLists and sent back to the component. For more
 * information about server commands, consult the game server code
 * at http://code.google.com/p/app-inventor-for-android/
 * Games instances are uniquely determined by a game id and an
 * instance id. In general, each App Inventor program should have
 * its own game id. Then, when running different instances of that
 * program, new instance ids should be used. Players are
 * represented uniquely by the email address registered to their
 * phones.
 * All call functions perform POSTs to a web server. Upon successful
 * completion of these POST requests, FunctionCompleted will be
 * triggered with the function name as an argument. If the post
 * fails, WebServiceError will trigger with the function name and the
 * error message as arguments. These calls allow for application
 * creators to deal with web service failures and keep track of the
 * success or failure of their operations. The only exception to this
 * is when the return value from the server has the incorrect game id
or instance id. In this case, the response is completely ignored and neither of these events will trigger.

@designerComponent(
    description = "Provides a way for applications to communicate with online game servers",
    category = ComponentCategory.EXPERIMENTAL)
@SimpleObject
@UsesPermissions(
    permissionNames = "android.permission.INTERNET, com.google.android.googleapps.permission.GOOGLE_AUTH")
public class GameClient implements OnResumeListener, OnStopListener {

    private static final String LOGTAG = "GameClient";

    // Parameter keys
    private static final String GAME_ID_KEY = "gid";
    private static final String INSTANCE_ID_KEY = "iid";
    private static final String PLAYER_ID_KEY = "pid";
    private static final String INVITEE_KEY = "inv";
    private static final String LEADER_KEY = "leader";
    private static final String COUNT_KEY = "count";
    private static final String TYPE_KEY = "type";
    private static final String INSTANCE_PUBLIC_KEY = "makepublic";
    private static final String MESSAGE_RECIPIENTS_KEY = "mrec";
    private static final String MESSAGE_CONTENT_KEY = "contents";
    private static final String MESSAGE_TIME_KEY = "mtime";
    private static final String MESSAGE_SENDER_KEY = "msender";
    private static final String COMMAND_TYPE_KEY = "command";
    private static final String COMMAND_ARGUMENTS_KEY = "args";
    private static final String SERVER_RETURN_VALUE_KEY = "response";
    private static final String MESSAGES_LIST_KEY = "messages";
    private static final String ERROR_RESPONSE_KEY = "e";
    private static final String PUBLIC_LIST_KEY = "public";
    private static final String JOINED_LIST_KEY = "joined";
    private static final String INVITED_LIST_KEY = "invited";
    private static final String PLAYERS_LIST_KEY = "players";

    // Command keys
    private static final String GET_INSTANCE_LISTS_COMMAND = "getinstancelists";
    private static final String GET_MESSAGES_COMMAND = "messages";
    private static final String INVITE_COMMAND = "invite";
    private static final String JOIN_INSTANCE_COMMAND = "joininstance";
    private static final String LEAVE_INSTANCE_COMMAND = "leaveinstance"
    private static final String NEW_INSTANCE_COMMAND = "newinstance";
    private static final String NEW_MESSAGE_COMMAND = "newmessage";
    private static final String SERVER_COMMAND = "servercommand";
    private static final String SET_LEADER_COMMAND = "setleader";
// URL for accessing the game server
private String serviceUri;
private String gameId;
private GameInstance instance;
private Handler androidUIHandler;
private Activity activityContext;

private String userEmailAddress = "";

// Game instances in the current GameId that this player has joined
private List<String> joinedInstances;
// Game instances to which this player has been invited
private List<String> invitedInstances;
// Game instances which have been made public.
private List<String> publicInstances;

/**
 * Creates a new GameClient component.
 * @param container the Form that this component is contained in.
 */
public GameClient(ComponentContainer container) {
    // Note that although this is creating a new Handler there is
    // only one UI thread in an Android app and posting to this
    // handler queues up a Runnable for execution on that thread.
    androidUIHandler = new Handler();
    activityContext = container.$context();
    Form form = container.$form();
    form.registerForOnResume(this);
    form.registerForOnStop(this);
    gameId = "";
    instance = new GameInstance("");;
    joinedInstances = Lists.newArrayList();
    invitedInstances = Lists.newArrayList();
    publicInstances = Lists.newArrayList();
    serviceUrl = "http://appinvgameserver.appspot.com";

    // This needs to be done in a separate thread since it uses
    // a blocking service to complete and will cause the UI to hang
    // if it happens in the constructor.
    AsynchUtil.runAsynchronously(new Runnable() {
        @Override
        public void run() {
            userEmailAddress = LoginServiceUtil.getPhoneEmailAddress(activityContext);
            if (!userEmailAddress.equals("")) {
                UserEmailAddressSet(userEmailAddress);
            }
    });
}
// Properties

/**
 * Returns a string indicating the game name for this application.
 * The same game ID can have one or more game instances.
 */
@SimpleProperty
public String GameId() {
    return gameId;
}

/**
 * Specifies a string indicating the family of the current game
 * instance. The same game ID can have one or more game instance
 * IDs.
 */
// Only exposed in the designer to enforce that each GameClient
// instance should be made for a single GameId.
@DesignerProperty(
    editorType = DesignerProperty.PROPERTY_TYPE_STRING,
    defaultValue = "\"")
public void GameId(String id) {
    this.gameId = id;
}

/**
 * Returns the game instance id. Taken together, the game ID and
 * the instance ID uniquely identify the game.
 */
@SimpleProperty
public String InstanceId() {
    return instance.getInstanceId();
}

/**
 * Returns the set of game instances to which this player has been
 * invited but has not yet joined. To ensure current values are
 * returned, first invoke @link #GetInstanceLists).
 */
@SimpleProperty
public List<String> InvitedInstances() {
    return invitedInstances;
}

/**
 * Returns the set of game instances in which this player is
 * participating. To ensure current values are returned, first
 * invoke @link #GetInstanceLists).
 */
@SimpleProperty
public List<String> JoinedInstances() {
    return joinedInstances;
}
/**
 * Returns the game’s leader. At any time, each game instance has
 * only one leader, but the leader may change with time.
 * Initially, the leader is the game instance creator. Application
 * writers determine special properties of the leader. The leader
 * value is updated each time a successful communication is made
 * with the server.
 * */

@SimpleProperty
public String Leader() {
    return instance.getLeader();
}

/**
 * Returns the current set of players for this game instance. Each
 * player is designated by an email address, which is a string. The
 * list of players is updated each time a successful communication
 * is made with the game server.
 * */

@SimpleProperty
public List<String> Players() {
    return instance.getPlayers();
}

/**
 * Returns the set of game instances that have been marked public.
 * To ensure current values are returned, first
 * invoke (link #GetInstanceLists).
 * */

@SimpleProperty
public List<String> PublicInstances() {
    return publicInstances;
}

/**
 * The URL of the game server.
 * */

@SimpleProperty
public String ServiceUrl() {
    return serviceUrl;
}

/**
 * Set the URL of the game server.
 * @param url The URL (include initial http://).
 * */

@DesignerProperty(
    editorType = DesignerProperty.PROPERTY_TYPE_STRING,
    defaultValue = "http://appinvgameserver.appspot.com")
public void ServiceURL(String url){
    if (url.endsWith("/")) {
        this.serviceUrl = url.substring(0, url.length() - 1);
else {
    this.serviceUrl = url;
}

/**
 * Returns the registered email address that is being used as the
 * player id for this game client.
 */
@SimpleProperty
public String UserEmailAddress() {
    if (userEmailAddress.equals("")) {
        Info("User email address is empty.");
    }
    return userEmailAddress;
}

/**
 * Changes the player of this game by changing the email address
 * used to communicate with the server.
 * This should only be used during development. Games should not
 * allow players to set their own email address.
 *
 * @param emailAddress The email address to set the current player
 * id to.
 */
@SimpleProperty
public void UserEmailAddress(String emailAddress) {
    userEmailAddress = emailAddress;
    UserEmailAddressSet(emailAddress);
}

/**
 * Indicates that a server request from a function call has
 * completed. This can be used to control a polling loop or
 * otherwise respond to server request completions.
 *
 * @param functionName The name of the App Inventor function that
 * finished.
 */
@SimpleEvent (description = "Indicates that a function call completed.")
public void FunctionCompleted(final String functionName) {
    androidUIHandler.post(new Runnable() {
        public void run() {
            Log.d(LOG_TAG, "Request completed: " + functionName);
            EventDispatcher.dispatchEvent(GameClient.this, "FunctionCompleted", functionName);
        }
    });
}
/**
 * Default Initialize event handler. Ensures that the GameId was
 * set by the game creator.
 */

@SimpleEvent
public void Initialize() {
    Log.d(LOG_TAG, "Initialize");
    if (gameId.equals("")) {
        throw new YailRuntimeError("Game Id must not be empty.", "GameClient Configuration Error.");
    }
    EventDispatcher.dispatchEvent(this, "Initialize");
}

/**
 * Indicates that a GetMessages call received a message. This could
 * be invoked multiple times for a single call to GetMessages.
 *
 * @param type The type of the message received.
 * @param contents The message’s contents. Consists of a list
 * nested to arbitrary depth that includes string, boolean and
 * number values.
 */

@SimpleEvent(description = "Indicates that a new message has " +
    "been received.")
public void GotMessage(final String type, final String sender, final List<Object> contents) {
    Log.d(LOG_TAG, "Got message of type " + type);
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "GotMessage", type, sender, contents);
        }
    });
}

/**
 * Indicates that InstanceId has changed due to the creation of a
 * new instance or setting the InstanceId.
 *
 * @param instanceId The id of the instance the player is now in.
 */

@SimpleEvent(description = "Indicates that the InstanceId " +
    "property has hanged as a result of calling " +
    "MakeNewInstance or SetInstance.")
public void InstanceIdChanged(final String instanceId) {
    Log.d(LOG_TAG, "InstanceId changed to " + instanceId);
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "InstanceIdChanged", instanceId);
        }
    });
}
/**
 * Indicates a user has been invited to this game instance by
 * another player.
 *
 * @param instanceId The id of the new game instance.
 */

@SimpleEvent(description = "Indicates that a user has been invited to " +
"this game instance.")

public void Invited(final String instanceId) {
    Log.d(LOG_TAG, "Player invited to " + instanceId);
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "Invited",
            instanceId);
        }
    });
}

/**
 * Indicates this game instance has a new leader. This could happen
 * in response to a call to SetLeader or by the side effects of a
 * server command performed by any player in the game.
 *
 * Since the current leader is sent back with every server
 * response, NewLeader can trigger after making any server call.
 *
 * @param playerId The email address of the new leader.
 */

@SimpleEvent(description = "Indicates that this game has a new " +
"leader as specified through SetLeader")

public void NewLeader(final String playerId) {
    androidUIHandler.post(new Runnable() {
        public void run() {
            Log.d(LOG_TAG, "Leader change to " + playerId);
            EventDispatcher.dispatchEvent(GameClient.this, "NewLeader",
            playerId);
        }
    });
}

/**
 * Indicates this game instance was created as specified via
 * MakeNewInstance. The creating player is automatically the leader
 * of the instance and the Instanceld property has already been set
 * to this new instance.
 *
 * @param instanceId The id of the newly created game instance.
 */

@SimpleEvent(description = "Indicates that a new instance was " +
"successfully created after calling MakeNewInstance.")

public void NewInstanceMade(final String instanceId) {
    androidUIHandler.post(new Runnable() {
        public void run() {
            Log.d(LOG_TAG, "New instance made: " + instanceId);
        }
    });
}
EventDispatcher.dispatchEvent(GameClient.this, "NewInstanceMade", instanceId);
}

/**
 * Indicates that a player has joined this game instance.
 * @param playerId The email address of the new player.
 */
@SimpleEvent(description = "Indicates that a new player has " +
"joined this game instance.")
public void PlayerJoined(final String playerId) {
    androidUIHandler.post(new Runnable() {
        public void run () {
            if (!playerId.equals(UserEmailAddress())) {
                Log.d(LOGTAG, "Player joined: "+ playerId);
                EventDispatcher.dispatchEvent(GameClient.this, "PlayerJoined",
playerId);
            }
        }
    });
}

/**
 * Indicates that a player has left this game instance.
 * @param playerId The email address of the player that left.
 */
@SimpleEvent(description = "Indicates that a player has left " +
"this game instance.")
public void PlayerLeft(final String playerId) {
    androidUIHandler.post(new Runnable() {
        public void run() {
            Log.d(LOGTAG, "Player left: "+ playerId);
            EventDispatcher.dispatchEvent(GameClient.this, "PlayerLeft",
playerId);
        }
    });
}

/**
 * Indicates that an attempt to complete a server command failed on
 * the server.
 * @param command The command requested.
 * @param arguments The arguments sent to the command.
 */
@SimpleEvent(
    description = "Indicates that a server command failed.")
public void ServerCommandFailure(final String command, final
YailList arguments) {
    androidUIHandler.post(new Runnable() {
        public void run() {
            Log.d(LOG_TAG, "Server command failed: "+ command);
            EventDispatcher.dispatchEvent(GameClient.this, "ServerCommandFailure",
command, arguments);
}}};
/** Indicates that a ServerCommand completed. */
*  *
* @param command The key for the command that resulted in this response.
* @param response The server response. This consists of a list nested to arbitrary depth that includes string, boolean and number values.
*/
@SimpleEvent(description = "Indicates that a server command returned successfully.")
public void ServerCommandSuccess(final String command, final List<Object> response) {
    Log.d(LOG_TAG, command + " server command returned.");
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "ServerCommandSuccess", command, response);
        }
    });
}

/** Indicates that the user email address property has been successfully set. This event should be used to initialize any web service functions. *
*  *
* This separate event was required because the email address was unable to be first fetched from the the UI thread without causing programs to hang. GameClient will now start fetching the user email address in its constructor and trigger this event when it finishes.
*/
@SimpleEvent(description = "Indicates that the user email address has been set.")
public void UserEmailAddressSet(final String emailAddress) {
    Log.d(LOGTAG, "Email address set.");
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "UserEmailAddressSet", emailAddress);
        }
    });
}

// Message events

/** Indicates that something has occurred which the player should be somehow informed of. *
*  *
* @param message the message.
*/
public void Info(final String message) {
    Log.d(LOG_TAG, "Info: " + message);
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "Info", message);
        }
    });
}

public void WebServiceError(final String functionName, final String message) {
    Log.e(LOGTAG, "WebServiceError: " + message);
    androidUIHandler.post(new Runnable() {
        public void run() {
            EventDispatcher.dispatchEvent(GameClient.this, "WebServiceError", functionName, message);
        }
    });
}

public void GetInstanceLists() {
    AsynchUtil.runAsynchronously(new Runnable() {
        public void run() {
            postGetInstanceLists();
        }
    });
}

private void postGetInstanceLists() {
    AsyncCallbackPair<JSONObject> readMessagesCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            // Process the response
        }
    };

}
processInstanceLists(response);

FunctionCompleted("GetInstanceLists");

public void onFailure(final String message) {
    WebServiceError("GetInstanceLists", "Failed to get up to date instance lists.");
}

public void processInstanceLists(JSONObject instanceLists) {
    try {
        JSONArray joinedList = JsonUtil.getStringListFromJSONArray(instanceLists,
            getJSONArray(JOINED_LIST_KEY));
        joinedInstances = JsonUtil.getStringListFromJSONArray(instanceLists,
            getJSONArray(PUBLIC_LIST_KEY));
        List<String> receivedInstancesInvited = JsonUtil.getStringListFromJSONArray(instanceLists,
            getJSONArray(INVITED_LIST_KEY));

        if (!receivedInstancesInvited.equals(InvitedInstances())) {
            List<String> oldList = invitedInstances;
            invitedInstances = receivedInstancesInvited;
            List<String> newInvites = new ArrayList<String>(receivedInstancesInvited);
            newInvites.removeAll(oldList);

            for (final String instanceInvited : newInvites) {
                Invited(instanceInvited);
            }
        }
    } catch (JSONException e) {
        Log.w(LOGTAG, e);
        Info("Instance lists failed to parse.");
    }
}

/**
 * Retrieves messages of the specified type.
 * Requests that only messages which have not been seen during
the current session are returned. Messages will be processed
in chronological order with the oldest first, however, only
the count newest messages will be retrieved. This means that
one could "miss out" on some messages if they request less than
the number of messages created since the last request for
that message type.

* Setting type to the empty string will fetch all message types.
* Even though those message types were not specifically requested,
* their most recent message time will be updated. This keeps
* players from receiving the same message again if they later
* request the specific message type.
* Note that the message receive times are not updated until after
* the messages are actually received. Thus, if multiple message
* requests are made before the previous ones return, they could
* send stale time values and thus receive the same messages more
* than once. To avoid this, application creators should wait for
* the get messages function to return before calling it again.

@param type The type of message to retrieve. If the empty string
is used as the message type then all message types will be
requested.
@param count The maximum number of messages to retrieve. This
should be an integer from 1 to 1000.

/**
 * @SimpleFunction(description = "Retrieves messages of the specified type.")
 */
public void GetMessages(final String type, final int count) {
    AsynchUtil.runAsynchronously(new Runnable() {
        public void run() { postGetMessages(type, count); }});
}

private void postGetMessages(final String requestedType, final int count) {
    AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<
            JSONObject>() {
        public void onSuccess(final JSONObject result) {
            try {
                int count = result.getInt(COUNT_KEY);
                JSONArray messages = result.getJSONArray(MESSAGES_LIST_KEY);
                for (int i = 0; i < count; i++) {
                    JSONObject message = messages.getJSONObject(i);
                    String type = message.getString(TYPE_KEY);
                    String sender = message.getString(MESSAGE_SENDER_KEY);
                    String time = message.getString(MESSAGE_TIME_KEY);
                    List<Object> contents = JsonUtil.getListFromJsonArray(
                        message.
                            getJSONArray(MESSAGE_CONTENT_KEY));
                    // Assumes that the server is going to return messages in
                    // chronological order.
                    if (requestedType.equals("")) {
                        instance.putMessageTime(requestedType, time);
                    }
                }
            }
        }
    });
}
instance.putMessageTime(type, time);
GotMessage(type, sender, contents);
}
} catch (JSONException e) {
    Log.w(LOG_TAG, e);
    Info("Failed to parse messages response.");
} FunctionCompleted("GetMessages");
}

public void onFailure(String message) {
    WebServiceError("GetMessages", message);
}

if (InstanceId().equals("")) {
    Info("You must join an instance before attempting to fetch messages.");
    return;
}

postCommandToGameServer(GETMESSAGESCOMMAND,
    Lists.<NameValuePair>newArrayList(
        new BasicNameValuePair(GAME_IDKEY, GameId()),
        new BasicNameValuePair(INSTANCEIDKEY, InstanceIdo),
        new BasicNameValuePair(PLAYERIDKEY, UserEmailAddress()),
        new BasicNameValuePair(COUNTKEY, new Integer(count).toString()),
        new BasicNameValuePair(MESSAGETIMEKEY, instance.getMessageTime(requestedType)),
        new BasicNameValuePair(TYPEKEY, requestedType)),
    myCallback);

/**
 * Invites a player to this game instance.
 * Players implicitly accept invitations when they join games by setting the instance id in their GameClient.
 * Invitations remain active as long as the game instance exists.
 * @param playerEmail a string containing the email address of the player to become leader. The email should be in one of the following formats:<br>"Name O. Person &ltname.o.person@gmail.com&gt"<br>"name.o.person@gmail.com".
 */
@SimpleFunction(
    description = "Invites a player to this game instance.")
public void Invite(final String playerEmail) {
    AsynchUtil.runAsynchronously(new Runnable() {
        public void run() { postInvite(playerEmail); }});
private void postInvite(final String inviteeEmail) {
    AsyncCallbackPair<JSONObject> inviteCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            try {
                String invitedPlayer = response.getString(INVITEE_KEY);
                if (invitedPlayer.equals("")) {
                    Info(invitedPlayer + " was already invited.");
                } else {
                    Info("Successfully invited " + invitedPlayer + ".");
                }
            } catch (JSONException e) {
                Log.w(LOG_TAG, e);
                Info("Failed to parse invite player response.");
            }
            FunctionCompleted("Invite");
        }
        public void onFailure(final String message) {
            WebServiceError("Invite", message);
        }
    };
    if (InstanceId().equals("")) {
        Info("You must have joined an instance before you can invite new players.");
        return;
    }
    postCommandToGameServer(INVITECOMMAND,
            Lists.<NameValuePair>newArrayList(
                new BasicNameValuePair(GAME_ID_KEY, GameId()),
                new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
                new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddresso),
                new BasicNameValuePair(INVITEE_KEY, inviteeEmail)),
            inviteCallback);
}

/**
 * Requests to leave the current instance. If the player is the
 * current leader, the lead will be passed to another player.
 * *
 * If there are no other players left in the instance after the
 * current player leaves, the instance will become unjoinable.
 * *
 * Upon successful completion of this command, the instance
 * lists will be updated and InstanceId will be set back to the
 * empty string.
 * *
 * Note that while this call does clear the leader and player
 * lists, no NewLeader or PlayerLeft events are raised.
 * */
@SimpleFunction(description = "Leaves the current instance.")
public void LeaveInstance() {
AsynchUtil.runAsynchronously(new Runnable() {
    public void run() {
        postLeaveInstance();
    }
});

private void postLeaveInstance() {
    AsyncCallbackPair<JSONObject> setInstanceCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            SetInstance(""); 
            processInstanceLists(response);
            FunctionCompleted("LeaveInstance");
        }
        public void onFailure(final String message) {
            WebServiceError("LeaveInstance", message);
        }
    };
    postCommandToGameServer(LEAVE_INSTANCE_COMMAND,
        Lists.<NameValuePair>newArrayList(
            new BasicNameValuePair(GAME_ID_KEY, GameId()),
            new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
            new BasicNameValuePair(USER_ID_KEY, UserEmailAddress()))
    ,
    setInstanceCallback);
}

/**  
 * Creates a new game instance. The instance has a unique  
 * instanceId, and the leader is the player who created it. The  
 * player that creates the game automatically joins it without  
 * being sent an invitation.  
 *  
 * The actual instance id could differ from the instanceId  
 * specified because the game server will enforce uniqueness. The  
 * actual instanceId will be provided to AppInventor when a  
 * NewInstanceMade event triggers upon successful completion of  
 * this server request.  
 *  
 * @param instanceId A string to use as for the instance  
 * id. If no other instance exists with this id, the new instance  
 * will have this id. However, since the id must be unique, if  
 * another instance exists with the same one, then a number  
 * will be appended to the end of this prefix.  
 * @param makePublic A boolean indicating whether or not the  
 * instance should be publicly viewable and able to be joined by  
 * anyone.  
 * */
@SimpleFunction(description = "Asks the server to create a new " +  
    "instance of this game.")
public void MakeNewInstance(final String instanceId, final boolean  
    makePublic) {
AsyncUtil.runAsynchronously(new Runnable() {
    public void run() { postMakeNewInstance(instanceId, makePublic); }});

private void postMakeNewInstance(final String requestedInstanceId,
    final Boolean makePublic) {
    AsyncCallbackPair<JSONObject> makeNewGameCallback = new
    AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            processInstanceLists(response);
            NewInstanceMade(InstanceId());
            FunctionCompleted("MakeNewInstance");
        }
        public void onFailure(final String message) {
            WebServiceError("MakeNewInstance", message);
        }
    };

    postCommandToGameServer(NEW_INSTANCE_COMMAND,
        Lists.<NameValuePair>newArrayList(
            new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
            new BasicNameValuePair(GAME_ID_KEY, GameId()),
            new BasicNameValuePair(INSTANCE_ID_KEY, 
                requestedInstanceId),
            new BasicNameValuePair(INSTANCE_PUBLIC_KEY, makePublic.
                toString()))),
        makeNewGameCallback, true);

    /**
     * Creates a new message and sends it to the stated recipients.
     * @param type A "key" for the message. This identifies the type of
     * message so that when other players receive the message they know
     * how to properly handle it.
     * @param recipients If set to an empty list, the server will send
     * this message with a blank set of recipients, meaning that all
     * players in the instance are able to retrieve it. To limit the
     * message receipt to a single person or a group of people,
     * recipients should be a list of the email addresses of the people
     * meant to receive the message. Each email should be in one of the
     * following formats:<br>
     * "Name O. Person &ltname.o.person@gmail.com&gt"<br>
     * "name.o.person@gmail.com"
     * @param contents the contents of the message. This can be any
     * AppInventor data value.
     */
    @SimpleFunction(description = "Sends a keyed message to all " +
        "recipients in the recipients list. The message will " +
        "consist of the contents list.")
    public void SendMessage(final String type, final YailList recipients
        , final YailList contents) {
        AsynchUtil.runAsynchronously(new Runnable() {
public void run() { postNewMessage(type, recipients, contents);
}};

private void postNewMessage(final String type, YailList recipients, YailList contents){
AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<JSONObject>(){
    public void onSuccess(final JSONObject response) {
        FunctionCompleted("SendMessage");
    }
    public void onFailure(final String message) {
        WebServiceError("SendMessage", message);
    }
};

if (InstanceIdo.equals("")) {
    Info("You must have joined an instance before you can send messages.");
    return;
}

postCommandToGameServer(NEWMESSAGE_COMMAND,
    Lists.<NameValuePair>newArrayList(
        new BasicNameValuePair(GAME_ID_KEY, GameId()),
        new BasicNameValuePair(INSTANCE_ID_KEY, InstanceIdo),
        new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
        new BasicNameValuePair(TYPEKEY, type),
        new BasicNameValuePair(MESSAGERECIPIENTSKEY, recipients.toJSONString()),
        new BasicNameValuePair(MESSAGECONTENTKEY, contents.toJSONString()),
        new BasicNameValuePair(MESSAGETIMEKEY, instance.getMessageTime(type))),
    myCallback);

/**
 * Submits a command to the game server. Server commands are
 * custom actions that are performed on the server. The arguments
 * required and return value of a server command depend on its
 * implementation.
 * For more information about server commands, consult the game
 * server code at:
 * http://code.google.com/p/app-inventor-for-android/
 */
@SimpleFunction(description = "Sends the specified command to " + 
"the game server.")
public void ServerCommand(final String command, final YailList arguments) {
    AsyncUtil.runAsynchronously(new Runnable() {
        public void run() { postServerCommand(command, arguments); }});
}

private void postServerCommand(final String command, final YailList arguments)
    AsyncCallbackPair<JSONObject> myCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject result) {
            try {
                ServerCommandSuccess(command, JsonUtil.getListFromJsonArray(result.
                    getJSONArray(MESSAGE_CONTENT_KEY)));
            } catch (JSONException e) {
                Log.w(LOGTAG, e);
                Info("Server command response failed to parse.");
                FunctionCompleted("ServerCommand");
            }
        } catch (JSONException e) {
            ServerCommandFailure(command, arguments);
            WebServiceError("ServerCommand", message);
        }
    });

    Log.d(LOG_TAG, "Going to post " + command + " with args " + arguments);
    postCommandToGameServer(SERVER_COMMAND,
        Lists.<NameValuePair>newArrayList(
            new BasicNameValuePair(GAME_ID_KEY, GameId()),
            new BasicNameValuePair(INSTANCE_ID_KEY, InstanceId()),
            new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
            new BasicNameValuePair(COMMANDTYPEKEY, command),
            new BasicNameValuePair(COMMANDARGUMENTSKEY, arguments.
                toJSONArray()),
            myCallback);
}

/**
 * Specifies the game instance id. Taken together, the game ID and
 * the instance ID uniquely identify the game.
 * @param instanceId the name of the game instance to join.
 */
@SimpleFunction(description = "Sets InstanceId and joins the "+
    "specified instance.")
public void SetInstance(final String instanceId) {
    AsyncUtil.runAsynchronously(new Runnable() {
        public void run() {
            if (instanceId.equals("")) {
                Log.d(LOG_TAG, "Instance id set to empty string.");
            }
if (!InstanceId().equals("")) {
    instance = new GameInstance("");
    InstanceIdChanged("");
    FunctionCompleted("SetInstance");
}
else {
    postSetInstance(instanceId);
}
}

private void postSetInstance(String instanceId) {
    AsyncCallbackPair<JSONObject> setInstanceCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            processInstanceLists(response);
            FunctionCompleted("SetInstance");
        }
        public void onFailure(final String message) {
            WebServiceError("SetInstance", message);
        }
    }
    AsyncCallbackPair<JSONObject> runAsynchronously = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            // Process response
        }
        public void onFailure(final String message) {
            // Handle failure
        }
    }
    postCommandToGameServer(JOIN_INSTANCE_COMMAND,
        Lists.<NameValuePair>newArrayList(
            new BasicNameValuePair(GAME_ID_KEY, GameId()),
            new BasicNameValuePair(INSTANCE_ID_KEY, instanceId),
            new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()))
        ,
        setInstanceCallback, true);
}

/**
 * Specifies the game’s leader. At any time, each game instance
 * has only one leader, but the leader may change over time.
 * Initially, the leader is the game instance creator. Application
 * inventors determine special properties of the leader.
 * The leader can only be set by the current leader of the game.
 * @param playerEmail a string containing the email address of the
 * player to become leader. The email should be in one of the
 * following formats:
 * <br>"name.0.person@gmail.com" <br>"name.o.person@gmail.com".
 * @simplefunction(description = "Tells the server to set the " +
 * "leader to playerid. Only the current leader may " +
 * "successfully set a new leader.")
 */
public void SetLeader(final String playerEmail) {
    AsynchUtil.runAsynchronously(new Runnable() {
        public void run() {
            postSetLeader(playerEmail);
        }
    });
private void postSetLeader(final String newLeader) {
    AsyncCallbackPair<JSONObject> setLeaderCallback =
    new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(final JSONObject response) {
            FunctionCompleted("SetLeader");
        }

        public void onFailure(final String message) {
            WebServiceError("SetLeader", message);
        }
    };

    if (InstanceIdo.equals("")) {
        Info("You must join an instance before attempting to set a
        leader.");
        return;
    }

    postCommandToGameServer(SET_LEADER_COMMAND,
    Lists.<NameValuePair>newArrayList(
        new BasicNameValuePair(GAME_ID_KEY, GameId()),
        new BasicNameValuePair(INSTANCE_ID_KEY, InstanceIdo),
        new BasicNameValuePair(PLAYER_ID_KEY, UserEmailAddress()),
        new BasicNameValuePair(LEADER_KEY, newLeader)),
    setLeaderCallback);
}

// Activity Lifecycle Management

/**
 * Called automatically by the operating system.
 * Currently does nothing.
 */
public void onResume() {
    Log.d(LOG_TAG, "Activity Resumed.");
}

/**
 * Called automatically by the operating system.
 * Currently does nothing.
 */
public void onStop() {
    Log.d(LOG_TAG, "Activity Stopped.");
}

// Utility Methods

private void postCommandToGameServer(final String commandName,
    List<NameValuePair> params, final AsyncCallbackPair<JSONObject>
    callback) {

postCommandToGameServer(commandName, params, callback, false);

private void postCommandToGameServer(String commandName,
        final List<NameValuePair> params, final AsyncCallbackPair<JSONObject> callback,
        boolean allowInstanceIdChange) {
    AsyncCallbackPair<JSONObject> thisCallback = new AsyncCallbackPair<JSONObject>() {
        public void onSuccess(JSONObject responseObject) {
            Log.d(LOGTAG, "Received response for " + commandName + ": " + responseObject.toString());
            try {
                if (responseObject.getBoolean(ERRORRESPONSEKEY)) {
                    callback.onFailure(responseObject.getString(SERVERRETURN_VALUE_KEY));
                } else {
                    String responseGameId = responseObject.getString(GAME_ID_KEY);
                    if (!responseGameId.equals(GameId())) {
                        Info("Incorrect game id in response: + " + responseGameId + ".");
                        return;
                    }
                    String responseInstanceId = responseObject.getString(INSTANCE_ID_KEY);
                    if (responseInstanceId.equals(InstanceId())) {
                        callback.onSuccess(responseObject.getJSONObject(SERVERRETURN.VALUE_KEY));
                    } else {
                        if (allowInstanceIdChange || InstanceId().equals("")) {
                            instance = new GameInstance(responseInstanceId);
                            updateInstanceInfo(responseObject);
                            InstanceIdChanged(responseInstanceId);
                        } else {
                            Info("Ignored server response to " + commandName + " for incorrect instance " + responseInstanceId + ".");
                            return;
                        }
                    }
                }
            } catch (JSONException e) {
                Log.w(LOGTAG, e);
            }
            callback.onFailure("Failed to parse JSON response to command " + commandName);
        }
    } catch (JSONException e) {
        Log.w(LOG_TAG, e);
        callback.onFailure("Failed to parse JSON response to command " + commandName);
    }
}
public void onFailure(String failureMessage) {
    Log.d(LOG_TAG, "Posting to server failed for " + commandName + " with arguments " +
    params + " \n Failure message: " + failureMessage);
    callback.onFailure(failureMessage);
}

WebServiceUtil.getInstance().postCommandReturningObject(ServiceUrl (), commandName, params,
    thisCallback);

private void updateInstanceInfo(JSONObject responseObject) throws
    JSONException {
    boolean newLeader = false;
    String leader = responseObject.getString(LEADER_KEY);
    List<String> receivedPlayers = JsonUtil.getStringListFromJsonArray
        (responseObject.
        getJSONArray(PLAYERS_LIST_KEY));

    if (!Leader().equals(leader)) {
        instance.setLeader(leader);
        newLeader = true;
    }

    PlayerListDelta playersDelta = instance.setPlayers(receivedPlayers );

    if (playersDelta != PlayerListDelta.NO_CHANGE) {
        for (final String player : playersDelta.getPlayersRemoved()) {
            PlayerLeft(player);
        }

        for (final String player : playersDelta.getPlayersAdded()) {
            PlayerJoined(player);
        }
    }

    if (newLeader) {
        NewLeader(Leader());
    }
}

B.2: GameInstance.java - A container for information pertaining to game instances.
import java.util.List;
import java.util.Map;

/**
 * A container for information about a GameInstance for use
 * with the App Inventor game framework.
 * @author billmag@google.com (Bill Magnuson)
 */
public class GameInstance {
    private String instanceId;
    private String leader;
    // players in the current game
    private List<String> players;
    // Use this to store the most recent time stamp of each message type received.
    private Map<String, String> messageTimes;

    /**
     * A GameInstance contains the most recent values
     * for the leader and players of a particular game instance.
     * This object is also used to keep track of the most recent
     * time that a particular message type was retrieved from the
     * server.
     * @param instanceId The unique String that identifies this
     * instance.
     */
    public GameInstance(String instanceId) {
        players = new ArrayList<String>(0);
        messageTimes = new HashMap<String, String>();
        this.instanceId = instanceId;
        this.leader = "";
    }

    /**
     * Return the instance id of this instance.
     * @return the instance id.
     */
    public String getInstanceId() {
        return instanceId;
    }

    /**
     * Return the current leader of this instance.
     * @return The email address of the current leader.
     */
    public String getLeader() {
        return leader;
    }
}
public void setLeader(String leader) {
    this.leader = leader;
}

public PlayerListDelta setPlayers(List<String> newPlayersList) {
    if (newPlayersList.equals(players)) {
        return PlayerListDelta.NOCHANGE;
    }
    List<String> removed = players;
    List<String> added = new ArrayList<String>(newPlayersList);
    players = new ArrayList<String>(newPlayersList);
    added.removeAll(removed);
    removed.removeAll(newPlayersList);
    // This happens if the players list is the same but the ordering
    // has changed for some reason.
    if (added.size() == 0 && removed.size() == 0) {
        return PlayerListDelta.NOCHANGE;
    }
    return new PlayerListDelta(removed, added);
}

public List<String> getPlayers() {
    return players;
}

/*
 * Return the list of players currently in this instance.
 * @return A list of the players in the instance.
 */
public List<String> getPlayers() {
    return players;
}
be used to filter available messages to find those that have not been received.

@param type The message type.
@return The most recently put value for this type.

public String getMessageTime(String type) {
    if (messageTimes.containsKey(type)) {
        return messageTimes.get(type);
    }
    return "";
}

/**
 * Puts a new time string for the specified message type.
 * The string should be some value that can be understood by its eventual consumer. It is left as a string here to remove the need to convert back and forth from Date/Time objects when dealing with web services.
 * @param type The message type.
 * @param time A string representing the time the message was created.
 */
public void putMessageTime(String type, String time) {
    messageTimes.put(type, time);
}

B.2 Utilities and Data Structures


package com.google.devtools.simple.runtime.components.util;
import com.google.devtools.simple.runtime.errors.YailRuntimeError;
import java.util.List;
import java.util.Collection;
import gnu.lists.FString;
import gnu.iists.LList;
import gnu.lists.Pair;
import org.json.JSONException;
import org.json.JSONObject;
The YailList is a wrapper around the gnu.list.Pair class used by the Kawa framework. YailList is the main list primitive used by App Inventor components.

@author gleitz@google.com (Benjamin Gleitzman)
@author billmag@google.com (Bill Magnuson)

public class YailList extends Pair {

// Component writers take note!
// If you want to pass back a list to the blocks language, the straightforward way to do this is simply to pass back an ArrayList. If you construct a YailList to return to codeblocks, you must guarantee that the elements of the list are "sanitized". That is, you must pass back a tree whose subtrees are themselves YailLists, and whose leaves are all legitimate Yail data types. See the definition of sanitization in runtime.scm.

/**
 * Create an empty YailList.
 */
public YailList() {
    super(YailConstants.YAIL_HEADER, LList.Empty);
}

private YailList(Object cdrval) {
    super(YailConstants.YAIL_HEADER, cdrval);
}

/**
 * Create a YailList from an array.
 */
public static YailList makeList(Object[] objects) {
    LList newCdr = Pair.makeList(objects, 0);
    return new YailList(newCdr);
}

/**
 * Create a YailList from a List.
 */
public static YailList makeList(List vals) {
    LList newCdr = Pair.makeList(vals);
    return new YailList(newCdr);
}

/**
 * Create a YailList from a Collection.
 */
public static YailList makeList(Collection vals) {
    LList newCdr = Pair.makeList(vals.toArray(), 0);
    return new YailList(newCdr);
}
/**
 * Return this YailList as an array.
 */

@override
public Object[] toArray() {
    if (cdr instanceof Pair) {
        return ((Pair) cdr).toArray();
    } else if (cdr instanceof LList) {
        return ((LList) cdr).toArray();
    } else {
        throw new YailRuntimeError("YailList cannot be represented as an array", "YailList Error.");
    }
}

/**
 * Return this YailList as an array of Strings.
 */

public String[] toStringArray() {
    int size = this.size();
    String[] objects = new String[size];
    for (int i = 1; i <= size; i++) {
        objects[i - 1] = String.valueOf(get(i));
    }
    return objects;
}

/**
 * Return a strictly syntactically correct JSON text representation of this YailList. Only supports String, Number,
 * Boolean, YailList, FString and arrays containing these types.
 */

public String toJSONString() {
    try {
        StringBuilder json = new StringBuilder();
        String separator = ""
        json.append('[');
        int size = this.size();
        for (int i = 1; i <= size; i++) {
            Object value = get(i);
            json.append(separator).append(getJsonRepresentation(value));
            separator = ",";
        }
        json.append(']');
        return json.toString();
    }
    catch (JSONException e) {
        throw new YailRuntimeError("List failed to convert to JSON.", "JSON Creation Error.");
    }
}
* Return the size of this YailList. */

@Override
public int size() {
    return super.size() - 1;
}

/**
 * Return a String representation of this YailList.
 */

@Override
public String toString() {
    if (cdr instanceof Pair) {
        return ((Pair) cdr).toString();
    } else if (cdr instanceof LList) {
        return ((LList) cdr).toString();
    } else {
        throw new RuntimeException("YailList cannot be represented as a String");
    }
}

/**
 * Return the String at the given index.
 */

public String getString(int index) {
    return (String) get(index + 1);
}

private String getJsonRepresentation(Object value) throws JSONException {
    if (value == null || value.equals(null)) {
        return "null";
    }
    if (value instanceof FString) {
        return JSONObject.quote(value.toString());
    }
    if (value instanceof YailList) {
        return ((YailList) value).toJSONString();
    }
    if (value instanceof Number) {
        return JSONObject.numberToString((Number) value);
    }
    if (value instanceof Boolean) {
        return value.toString();
    }
    if (value.getClass().isArray()) {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        String separator = " ";
        for (Object o: (Object[]) value) {
            sb.append(separator).append(getJsonRepresentation(o));
            separator = ",";
        }
    }
}
B.4: JsonUtil.java - Utility functions for converting JSON to data representations understood by App Inventor.

```java
package com.google.devtools.simple.runtime.components.util;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class JsonUtil {

    private JsonUtil() {
    }

    /**
     * Provides utility functions to create Java collections out of JSON.
     *
     * @param jArray The JSONArray to convert.
     * @return A List of the String representation of each item in the JSON array.
     * @throws JSONException if an element of jArray cannot be converted to a String.
     */

    public static List<String> jsonArrayToList(JSONArray jArray) throws JSONException {
        List<String> result = new ArrayList<String>;
        for (int i = 0; i < jArray.length(); i++) {
            if (jArray.get(i) instanceof JSONArray) {
                JSONArray nestedArray = (JSONArray) jArray.get(i);
                result.addAll(jsonArrayToList(nestedArray));
            } else if (jArray.get(i) instanceof JSONObject) {
                JSONObject nestedObject = (JSONObject) jArray.get(i);
                result.add(nestedObject.toString());
            } else {
                result.add(jArray.getString(i));
            }
        }
        return result;
    }
}
```
public static List<String> getStringListFromJsonArray(JSONArray jArray) throws JSONException {
    List<String> returnList = new ArrayList<String>();
    for (int i = 0; i < jArray.length(); i++) {
        String val = jArray.getString(i);
        returnList.add(val);
    }
    return returnList;
}

/**
 * Returns a Java Object list of a JSONArray with each item in
 * the array converted using convertJsonItem().
 * @param jArray The JSONArray to convert.
 * @return A List of Strings and more Object lists.
 * @throws JSONException if an element in jArray cannot be
 * converted properly.
 */
public static List<Object> getListFromJsonArray(JSONArray jArray) throws JSONException {
    List<Object> returnList = new ArrayList<Object>();
    for (int i = 0; i < jArray.length(); i++) {
        returnList.add(convertJsonItem(jArray.get(i)));
    }
    return returnList;
}

/**
 * Returns a list containing one two item list per key in jObject.
 * Each two item list has the key String as its first element and
 * the result of calling convertJsonItem() on its value as the
 * second element. The sub-lists in the returned list will appear
 * in alphabetical order by key.
 * @param jObject The JSONObject to convert.
 * @return A list of two item lists: [String key, Object value].
 * @throws JSONException if an element in jObject cannot be
 * converted properly.
 */
public static List<Object> getListFromJsonObject(JSONObject jObject) throws JSONException {
    List<Object> returnList = new ArrayList<Object>();
    Iterator<String> keys = jObject.keys();
    List<String> keysList = new ArrayList<String>();
    while (keys.hasNext()) {
        keysList.add(keys.next());
    }
    Collections.sort(keysList);
    for (String key : keysList) {
        List<Object> nestedList = new ArrayList<Object>();
        }
nestedList.add(key);
nestedList.add(convertJsonItem(jObject.get(key)));
returnList.add(nestedList);
}

return returnList;
}

/**
 * Returns a Java object representation of objects that are
 * encountered inside of JSON created using the org.json package.
 * JSON arrays and objects are transformed into their list
 * representations using getListFromJsonArray and
 * getListFromJsonObject respectively.
 * Java Boolean values and the Strings "true" and "false" (case
 * insensitive) are inserted as Booleans. Java Numbers are
 * inserted without modification and all other values are inserted
 * as their toString(). value.
 * @param o An item in a JSON array or JSON object to convert.
 * @return A Java Object representing o or the String "null"
 * @throws JSONException if o fails to parse.
 */
public static Object convertJsonItem(Object o) throws JSONException {
    if (o == null) {
        return "null";
    }

    if (o instanceof JSONObject) {
        return getListFromJsonObject((JSONObject) o);
    }

    if (o instanceof JSONArray) {
        return getListFromJsonArray((JSONArray) o);
    }

    if (o.equals(Boolean.FALSE) || (o instanceof String && ((String) o).equalsIgnoreCase("false"))) {
        return false;
    }

    if (o.equals(Boolean.TRUE) || (o instanceof String && ((String) o).equalsIgnoreCase("true"))) {
        return true;
    }

    if (o instanceof Number) {
        return o;
    }

    return o.toString();
package com.google.devtools.simple.runtime.components.android.util;

import android.util.Log;

import org.apache.http.NameValuePair;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.List;

import org.apache.http.conn.scheme.PlainSocketFactory;
import org.apache.http.conn.scheme.Scheme;
import org.apache.http.conn.scheme.SchemeRegistry;
import org.apache.http.conn.ssl.SSLSocketFactory;
import org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager;
import org.apache.http.protocol.HTTP;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

/**
 * These commands post to the Web and get responses that are assumed
 * to be JSON structures: a string, a JSON array, or a JSON object.
 * It's up to the caller of these routines to decide which version
 * to use, and to decode the response.
 *
 * @author halabelson@google.com (Hal Abelson)
 * @autho billmag@google.com (Bill Magnuson)
 */
public class WebServiceUtil {

private static final WebServiceUtil INSTANCE = new WebServiceUtil();
private static final String LOG_TAG = "WebServiceUtil";
private static HttpClient httpClient = null;
private static Object httpClientSynchronizer = new Object();

private WebServiceUtil() {
}

/**
 * Returns the one <code>WebServiceUtil</code> instance
 * @return the one <code>WebServiceUtil</code> instance
 */
public static WebServiceUtil getInstance() {
    synchronized(httpClientSynchronizer) {
        if (httpClient == null) {
            SchemeRegistry schemeRegistry = new SchemeRegistry();
            schemeRegistry.register(new Scheme("http", PlainSocketFactory.
                getSocketFactory(), 80));
            schemeRegistry.register(new Scheme("https", SSLSocketFactory.
                getSocketFactory(), 443));
            BasicHttpParams params = new BasicHttpParams();
            HttpConnectionParams.setConnectionTimeout(params, 20 * 1000);
            HttpConnectionParams.setSoTimeout(params, 20 * 1000);
            ConnManagerParams.setMaxTotalConnections(params, 20);
            ThreadSafeClientConnManager manager = new
                ThreadSafeClientConnManager(params, schemeRegistry);
            WebServiceUtil.httpClient = new DefaultHttpClient(manager, params);
        }
        return INSTANCE;
    }
}

/**
 * Make a post command to serviceURL with params and return the
 * response String as a JSON array.
 * @param serviceURL The URL of the server to post to.
 * @param commandName The path to the command.
 * @param params A List of NameValuePairs to send as parameters
 * with the post.
 * @param callback A callback function that accepts a JSON array
 * on success.
 */
public void postCommandReturningArray(String serviceURL, String
    commandName,
    List<NameValuePair> params, final AsyncCallbackPair<JSONArray>
    callback) {
    AsyncCallbackPair<String> thisCallback = new AsyncCallbackPair<String>() {
        public void onSuccess(String httpResponseString) {
            try {
                callback.onSuccess(new JSONArray(httpResponseString));
            } catch (JSONException e) {
                // Error handling...
            }
        }
    }
    thisCallback.onSuccess(new String[]{serviceURL, commandName, params.toString()});
}
catch (JSONException e) {
    callback.onFailure(e.getMessage());
}

public void onFailure(String failureMessage) {
    callback.onFailure(failureMessage);
}

postCommand(serviceURL, commandName, params, thisCallback);

/**
 * Make a post command to serviceURL with parameters and
 * return the response String as a JSON object.
 *
 * @param serviceURL The URL of the server to post to.
 * @param commandName The path to the command.
 * @param params A List of NameValuePairs to send as parameters
 * with the post.
 * @param callback A callback function that accepts a JSON object
 * on success.
 */
public void postCommandReturningObject(final String serviceURL, final
  String commandName, final List<NameValuePair> params, final AsyncCallbackPair<JSONObject>
callback) {
  AsyncCallbackPair<String> thisCallback = new AsyncCallbackPair<String>() {
    public void onSuccess(String httpResponseString) {
      try {
        callback.onSuccess(new JSONObject(httpResponseString));
      } catch (JSONException e) {
        callback.onFailure(e.getMessage());
      }
    }
    public void onFailure(String failureMessage) {
      callback.onFailure(failureMessage);
    }
  };  
  postCommand(serviceURL, commandName, params, thisCallback);

  /*
   * Make a post command to serviceURL with parameters and return the
   * response String.
   *
   * @param serviceURL The URL of the server to post to.
   * @param commandName The path to the command.
   * @param params A List of NameValuePairs to send as parameters
   * with the post.
   * @param callback A callback function that accepts a String on
   * success.
   */
public void postCommand(final String serviceURL, final String
commandName,
   List<NameValuePair> params, AsyncCallbackPair<String> callback)
{
   Log.d(LOG_TAG, "Posting " + commandName + " to " + serviceURL + "
with arguments " + params);

   if (serviceURL == null || serviceURL.equals("")) {
      callback.onFailure("No service url to post command to.");
   }

   final HttpPost httpPost = new HttpPost(serviceURL + "/" +
commandName);

   if (params == null) {
      params = new ArrayList<NameValuePair>();
   }

   try {
      String httpResponseString;
      ResponseHandler<String> responseHandler = new
           BasicResponseHandler();
      httpPost.setEntity(new UrlEncodedFormEntity(params,
HTTP.UTF_8))
          httpPost.setHeader("Accept", "application/json");
      httpResponseString = httpClient.execute(httpPost,
            responseHandler);
      callback.onSuccess(httpResponseString);
   } catch (UnsupportedEncodingException e) {
      Log.w(LOG_TAG, e);
      callback.onFailure("Failed to encode params for web service call").
   } catch (ClientProtocolException e) {
      Log.w(LOG_TAG, e);
      callback.onFailure("Communication with the web service
           encountered a protocol exception.");
   } catch (IOException e) {
      Log.w(LOG_TAG, e);
      callback.onFailure("Communication with the web service timed out
           ");
   }
}
Bibliography


