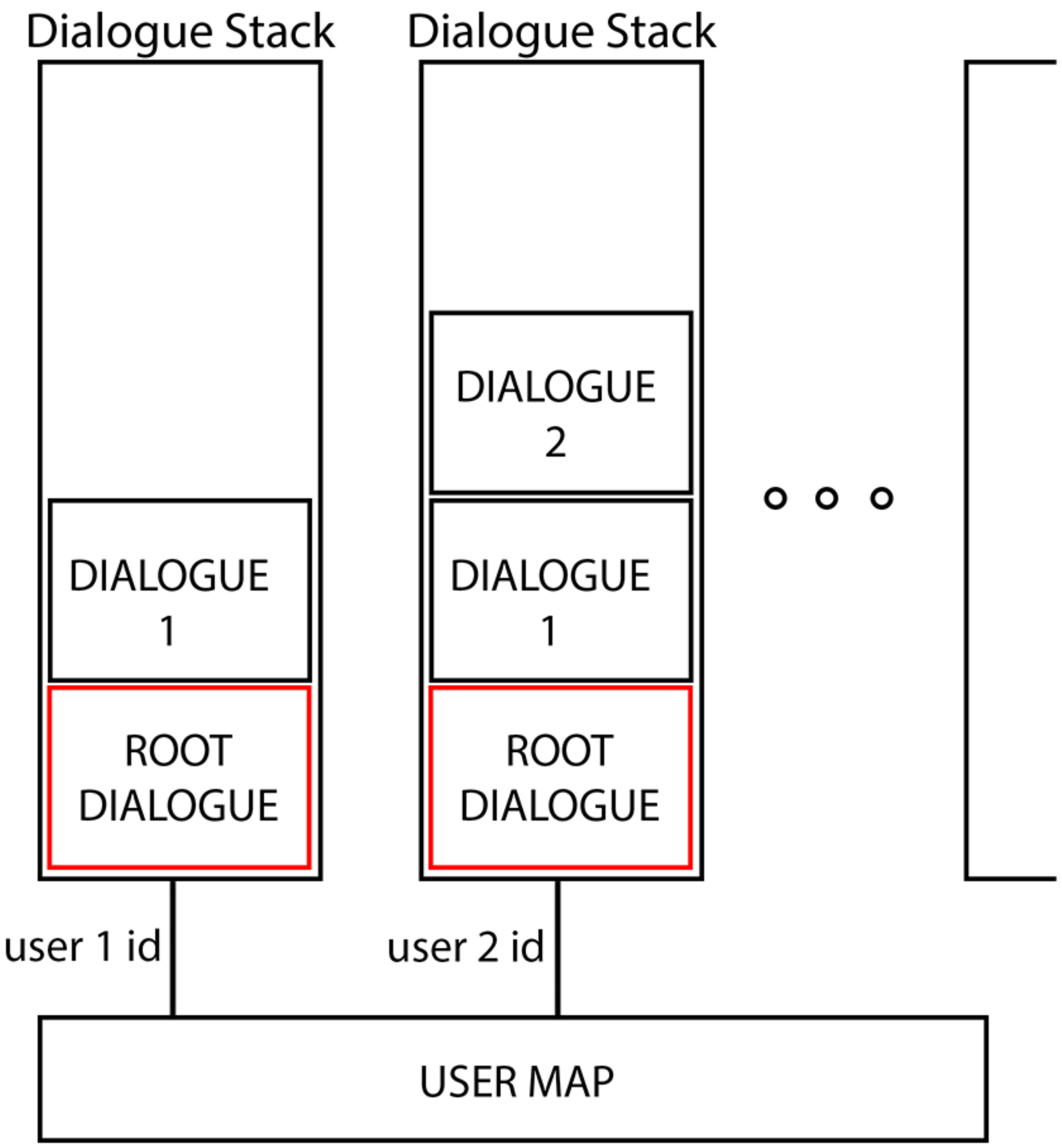
Dialogue Stacks

Author: Ian Wang

Overview

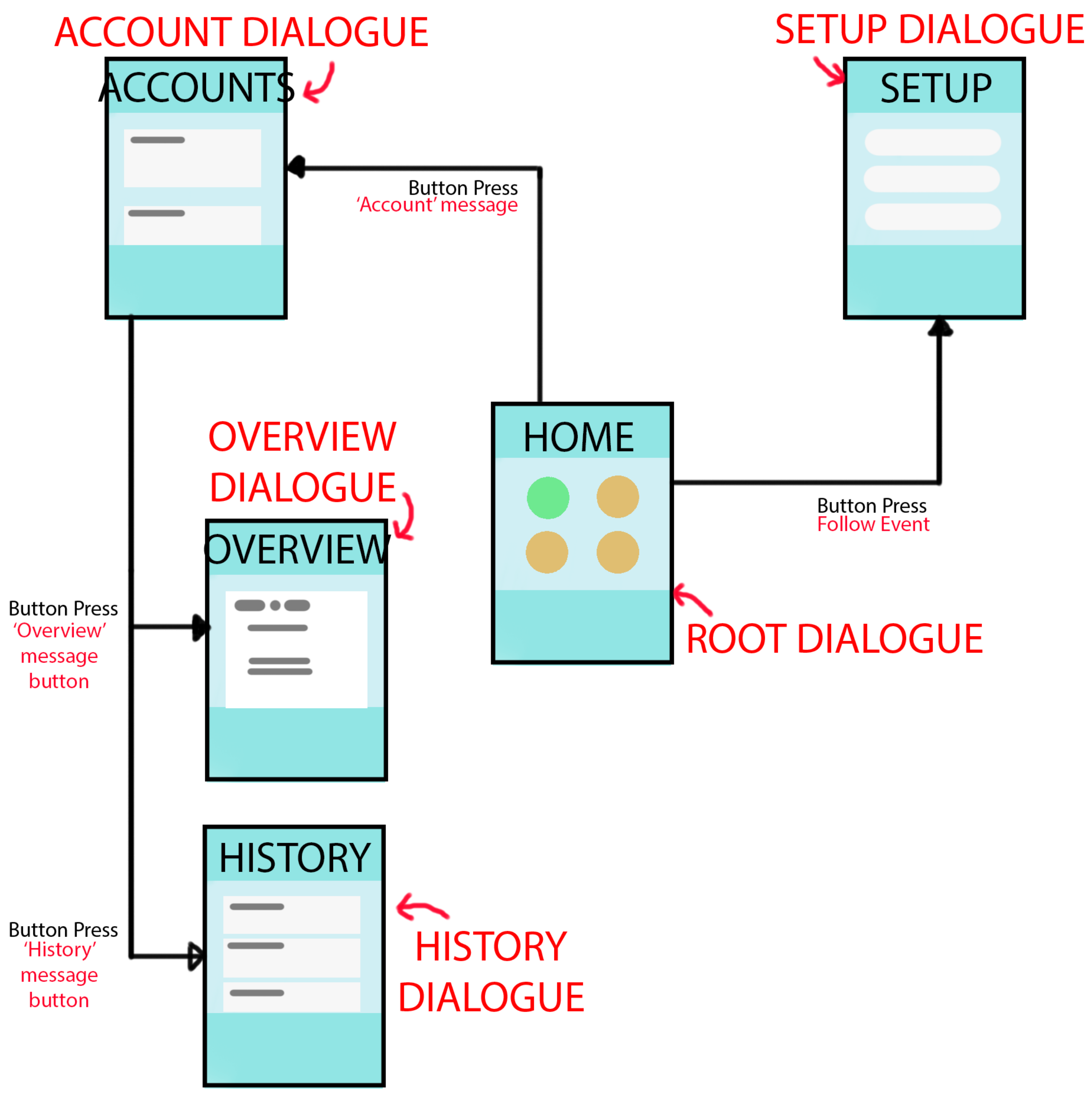
A more streamlined method of developing bot applications, called **dialogue stacks**, are implemented in the API as well. A *Dialogue* defines how the bot server handles the incoming events of a certain user, and each user is allocated their own dialogue stack. Dialogues can maniplate the stack from within itself, allowing extensive customization on the behaviour within each dialogue.

Each user is given a dialogue stack to keep track of their current dialogues. The top of the stack is the dialogue that the user is currently interacting with, so the *handleEvent* function being called for incoming webhooks is the handleEvent function of the top dialogue. Each dialogue can push/pop from the stack from within its own code, as the *Dialogue* class provides a push/pop function to work with.



Benefits

The biggest benefit of dialogue stacks is that it is easy to reason dialogue flow. Each dialogue can be thought of as a 'page' in the sense of a traditional app. Each dialogue handles the inputs differently, just like how each page has different functions for the user to interact with. The stack implementation allows user to easily return to a 'previous' dialogue, just like how using the back button in an app or website will lead you to where you were before. Once a traditional app flow diagram is drawn out, it is essentially trivial to convert it to a dialogue flow using dialogue stacks.



Another big benefit is the implicit separation of user states. Since each user gets their own dialogue stack, then we know that variables stored inside a dialogue is individual to each user. Thus, storing temporary data from users becomes as trivial as creating a new variable inside a dialogue to keep track of whatever it is that needs to be kept track of.

Issues

One concern that could be raised with the implementation of dialogue stacks is performace. Since each user is allocated a dialogue stack, the program has to allocate extra memory for each user as well as pass through some extra levels of indirection. However, in most cases, these concerns will not have too significant of an impact as long as the dialogue flow is well designed.

First, to address the issue of dialogue stacks requiring too much memory, a deeper look into the implementation of dialogue stacks is required. Each Dialogue contains 4 object references, which is 4 * 32 bits in most JVMs, which means that the dialogue object itself takes up 16 bytes of data. The dialogue stack itself also contains an object reference, which is another 32 bits (4 bytes) of data, so each user takes up at least 20 bytes of memory to store its dialogue stack. Since everything is stored in a Hashmap, another object reference is stored alongside the userId String. Since the userId is stored anyways, its cost will not be counted towards the cost of Dialogue Stacks. Thus, each user takes an extra 24 bytes of memory total at minimum.

If the bot server were to handle 1000 users, it would need roughly 24 kilobytes of memory to store the dialogue stacks of each user, which is easily handled by any modern machine. Even at 1,000,000 users, there would be 24 megabytes of memory needed to store all the dialogue stacks, which still isn't much for modern processors to handle. The problem comes when dialogue flow is badly designed and dialogue stacks start to contain too many dialogues. The average number of dialogues per stack is directly proportional to the memory used, so if there are on average 3 dialogues per stack, then we will require roughly 3 times the memory of one dialogue in each dialogue stack. Therefore, a well designed dialogue flow averaging one dialogue per user will not have a significant impact on the memory required to run the program.

Addressing the efficiency issues with the levels of indirection using dialogue stacks, only the implementation of the storing of dialogue stacks needs to be examined. Looking at the implementation without dialogue stacks, an incoming event goes through a central routing function to convert the JSON data into a Message object, and then calls the handleEvent function. With dialogue stacks, the program finds the dialogue stack of the corresponding user after converting the data into a Message object, and then calls the handleEvent function of the top dialogue of the stack.

One extra function call is small enough to not be noticable, so the biggest performance difference should be finding the dialogue stack of the user. Since a HashMap is used to store the users and their corresponding dialogue stacks, finding the correct stack in a map is constant time, the Map is big enough for there to be minimal collisions. Even with collisions, the HashMap will detect if the entry is big enough to justify using a Binary Search Tree, which will result in logarithmic time at worst.

As shown above, the issues of dialogue stacks don't have a significant impact on performance, and the trade-off is easily worth it for easier dialogue design and code readability.

Using Dialogue Stacks

The base **Dialogue** class has one main function to override, the **handleEvent** function.

```
// main event handler function
public abstract void handleEvent(WebHookEvent event, String userId);
```

The WebhookHandler triggers the overriden *handleEvent* function of the current top of the dialogue stack. From there, the dialogue can decide how to handle the incoming event, whether it's interacting with a remote API or if it wants to push another dialogue on the stack.

The **push** and **pop** functions allow the programmer to interact with the dialogue stack without being exposed to how it is implemented.

One important thing to note is that the Dialogue class contains a static **sender** variable to store a reference to the message sender that should be used to send push/reply messages. This means that the *setSender* function has to be called once when the program runs, from which point on any Dialogue subclass will be able to send messages via the sender.

```
public static void setSender(MessageSender sender)
```

A **Root Dialogue** is a Dialogue that is never removed from the stack, each user should have one and only one Root Dialogue at the base of its stack. In terms of implementation, there is only one difference between root dialogues and normal dialogues, and that is root dialogues require a **create** function. This is because the bot server needs to be able to instantiate a new root dialogue for each new user, and it does so by using the create function. Below is the code for the root dialogue class:

```
public abstract class RootDialogue extends Dialogue {

    // Create function for WebhookHandler to make as many root dialogues as it
needs
    public abstract RootDialogue create();

    public RootDialogue() {
        // constructor stuff
    }

    // Use a demo echo root dialogue as a default if not specified by user
    public static RootDialogue createDefault() {
        return new EchoRootDialogue();
    }
}
```

Example Dialogue Implementation

```java
public class RootDialogueDemo extends RootDialogue {
    // Returns a new root dialogue for new user dialgoue stacks
    @Override
    public RootDialogue create() {
        return new RootDialogueDemo();
    }
    @Override
    public void handleEvent(WebHookEvent event, String userId) {
        // Handle different event types differently via type property
        if (event.type() == WebHookEventType.MESSAGE) {
            MessageEvent messageEvent = (MessageEvent) event;
            if (messageEvent.message().type() == MessageType.TEXT) {
                TextMessage textMessage = (TextMessage) messageEvent.message();
                parseTextMessage(textMessage.getText(), userId);
            }
        }
        if (event.type() == WebHookEventType.FOLLOW) {
            FollowEvent followEvent = (FollowEvent) event;
            TextMessage msg = new TextMessage("Type setup to get started.");
            Util.sendSingleReply(sender, followEvent.replyToken(), msg);
        }
    }

    private void parseTextMessage(String text, String userId) {
        if (text.toLowerCase().contains("setup")) {
            // Example of pushing a new dialogue onto the stack
            // The next recieved event will call handleEvent of SetupDialogue
            push(new SetupDialogue(userId));
        } else if (text.toLowerCase().contains("accounts")) {
            handleAccountCommand(userId);
        } else if (text.toLowerCase().contains("info")) {
            // Dialogue can also handle logic
            UserData data = SomeRemoteAPI.getUserInfo(userId);
            Util.sendSingleTextPush(sender, userId, data.toString());
        } else {
            Util.sendSingleTextPush(sender, userId, "Message not understood");
        }
    }

    private void handleAccountCommand(String userId) {
        // Some account handling code here
    }
}
```

Dialogue Design Tips

Dialogue Behaviour

If there is a desired behavior for when a dialogue is added to the stack, then code can be added to the **constructor** so that it will run when dialogue is instantiated.

If there is a desired behavior for when a dialogue is returned to after another dialogue is popped (i.e. run some code when dialogue A becomes top of the stack after popping dialogue B, which was on top of dialogue A), then override the **recieve** function of the dialogue.

```
/**
 * Function to override if there is functionality desired for when an element is popped
 *      and the current dialogue is the next in the stack.
 */
protected void recieve();
```

If for some reasons there needs to be extra steps taken for the push/pop functions, they aren'y implemented as final methods so they can be overriden as well. Just be sure to call the push/pop method of the base class within the overriden method so that the intended behaviour of push/pop is still maintained.

Storing temporary state

If user state needs to be stored, it is as simple as adding a new variable to the subclassed Dialogue class. For example, if an *email query* dialogue wants to store the email while it does some other things, then an *email string* can be added to the dialogue. This works because each user has a unique dialogue stack, so variables within a dialogue are specific to a user, whereas usually the program would need to keep track of which users currently typed which email.

General Tips

Finally, it is a good idea to have a limit on the amount of dialogues that can be stored within a stack. In general, it should not cause any memory issues, since accessing the top of the stack is **O(1)** complexity. However, as explained in the *issues* section above, the memory cost can increase drastically if every user has many dialogues. If there is a case where a dialogue with high complexity is needed, consider alternative solutions to just adding a lot of dialogues on to the stack, such as having a base dialogue that interacts with multiple sub dialogues which return to the base dialogue once they are done, or using an API on another server etc.

References

- Microsoft Bot Builder Dialogues