Technical Guide

Niall Kelly (20461772) & Adam Gray (20364103)

Project Title: START Web Application

Supervisor: David Sinclair

Introduction

Overview

The START Web Application is web based IDE for our own START programming language. The IDE includes features such as syntax highlighting, debugging modes, memory mapping and a session saving feature. As well as this, the application allows for users to make their own accounts, to save their progress over multiple sessions, learning resources such as informative videos and START programming documents to help beginners learn to program, as START is a beginner programming language. The application is a Django-Python application,

with use of Java and ANTLR4 for use in the backend making up our programming language and debugger. The language is an interpreted language with a simple syntax, relying on key words rather than symbols to make it more beginner friendly. The frontend is designed to have minimal features and buttons to allow for a more smooth user experience. The frontend consists of HTML, CSS with Bootstrap and some vanilla JavaScript, as well as the use of Ace, a standalone code editor written in JavaScript we used as our browser based editor.

Glossary

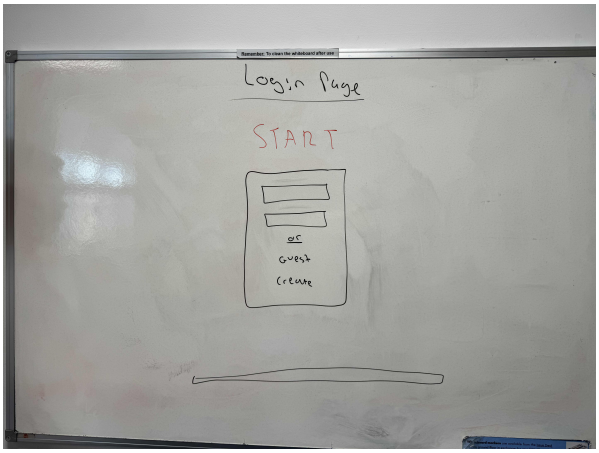
- START - START is our own programming language we developed in 2023. It is an interpreted language written in Java with ANTLR4.
- Django - Django is a Python framework that makes it easier to create web sites using Python.
- ANTLR4 - A powerful parser generator for reading, processing, executing, or translating structured text or binary files.
- Bootstrap - Bootstrap is the most popular CSS Framework for developing responsive and mobile-first websites.
- Jest - A JavaScript testing framework used for testing JavaScript code.
- Ace - Ace is a code editor written in JavaScript to be used as a browser based editor that can compete with other editors such as Vim or Eclipse in both features and usability. It can be easily embedded into a web page or JavaScript application.
- GitLab - A web-based Git repository that provides free open and private repositories.
- Docker - Docker provides the ability to package and run an application in a loosely isolated environment called a container.

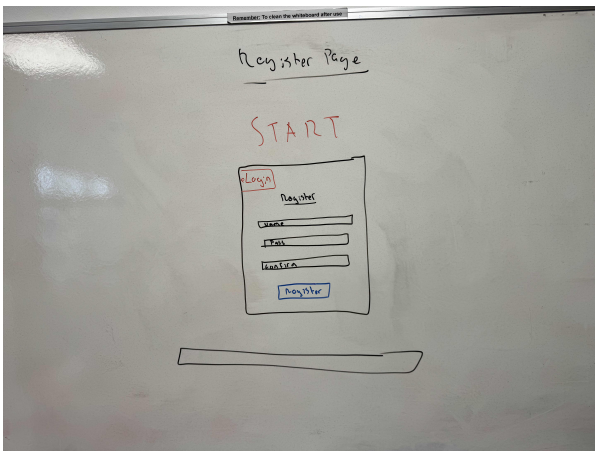
Motivation

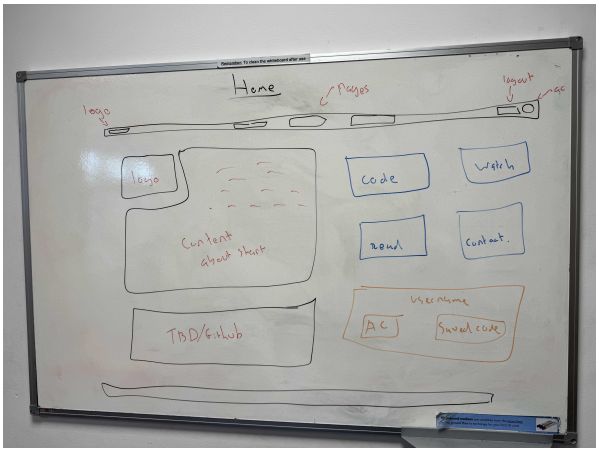
In 2023, our third year project was creating our own programming language called START, which was a beginner friendly programming language written in Java with ANTLR4. We were very happy with how it turned out, as it had high usability in solving beginner programming problems, while being easy to learn. During the

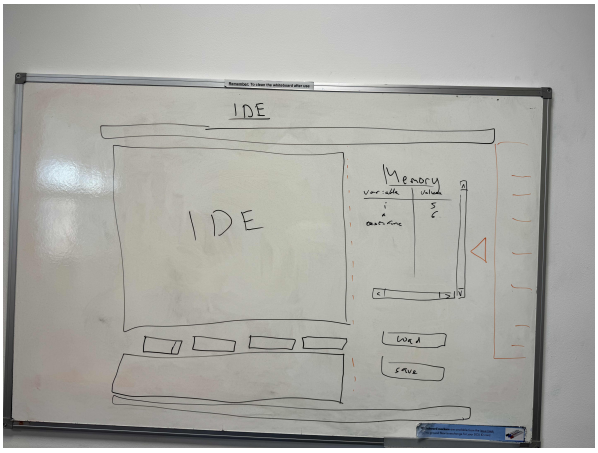
summer of 2023, we began to talk about how much we enjoyed our previous project, and wanted to see if there was something similar we could do for the next year, as we did not want to completely move away from our own language we had created. This is where we came to the realisation, our language was easy to use, but not so easy to access, so how could we solve this new problem, while providing more features to users? This is where we discussed creating a web IDE for our own language, much like we had seen during interviews using HackerRank. With this in mind we drew up some basic design ideas we could implement as part of our final design, which can be seen below.

Initial Mock-Ups of the Web UI









Pre-Development Discussion & Research

Before beginning our development of this project, we discussed some potential problems that would need to be discussed. We had a few questions we needed to answer before we could start:

1. What framework and language would we use for the web app?
2. How would we create the text editor?
3. What features should we include for the users?
4. Should we build upon START for this project?

1 - As we discussed, we used Python and Django to build our web app, but our decision to use these did not come unchallenged. When we originally discussed the idea, we had discussed potentially using a C# and TypeScript stack to build the app, however, we realised the app was really just the “shelf” holding up the “product”, where the product in question was our language and debugger. Because of this, we chose to use Python, as we had some limited experience before with it, and wished to extend our knowledge beyond what we learned during our undergraduate program.

2 - When it came to creating our own text editor, we originally had looked into how to make our own IDE from scratch using Javascript, and while this would be doable, it would take significant work, but we were ready to do this for a project we were passionate about. During our project proposal meeting, a member of DCU School of Computing Dr. Michael Scriney, suggested we look into finding an open source text editor, or even the VSCode open source code to give us a great starting point to allow us to focus on making the IDE perfected for our language. With this in mind, we found the Ace project on GitHub. This allowed us to save time later in the project developing the text editor, as we will discuss more later on.

3 - We wanted to keep our list of core features in the IDE limited, but powerful, as to not overwhelm the user, who will likely be a beginner programmer. We ended up coming up with a shortlist of features after looking at the main features of several IDE’s such as VSCode, Visual Studio and IntelliJ:

- Debug mode
- Breakpoints

- Memory map
- Problems to learn programming concepts
- User accounts to save progress

4 - When it came to our own language, we discussed if we should expand upon the features, however, this lead us to realise, beginners had all core features needed to solve simple beginner problems, which is what the language was designed for, and by adding easy ways to solve problems, we would be breaking our principle of Orthogonality. Instead, we then discussed potentially making a secondary version of the language, that instead of acting as it normally would, could act as the debugger, reacting to the input.

Design

The following sub-sections will discuss the architectural development of our web application by examining the main components of the front and backend of the application.

System Architecture

Language & Debugger Backend

Given we are using our previously developed language from last years project, a small description of its working and development would be appropriate. START uses ANTLR4, a 3rd party parser generator to read and process structured text files. However it needs to know the elements of our language to parse it, so we used a `.g4` file to lay out our grammar rules for START, an example being `assignment: NAME 'is' expression;`. Then once the grammar file was finished we created a visitor, used to generator rules for the parser to follow, which we would later override to create our language rules. We then used a driver file name `start.java` to take an input text file, parse it, and call the visitor on the text to determine what the language should do. When the visitor locates the next rule to visit, it calls upon the relevant function which was generated by ANTLR. Then in our `startMainVisitor` we override this call to allow us to manipulate the text in a desired way, hence, how the language works, an example function is seen below:

```
@Override
public Object visitFunction(startParser.FunctionContext ctx) {
    //look at the map on the top of the stack
    HashMap<String, Object> map = mappy.peek();

    //create a new arraylist to store the arguments
    ArrayList<String> args = new ArrayList<String>();
    //visit the arguments and add them to the arraylist
    for(int i = 1; i < ctx.NAME().size(); i++){
        args.add(ctx.NAME(i).getText());
    };
    //create a new function object
    Function func = new Function(args, ctx.block());
    //print out the block
    map.put(ctx.NAME(0).getText(), func);
    return null;
}
```

In the above function, the `@Override` will override the original template in the generated visitor, and in this example, will store the arguments of a function found into an array, then keep the function object of arguments and code stored in the map. In essence, this is how the rules of the language operate.

With this in mind, we knew now we needed a way to implement breakpoints into our language for the debugger to be able to step through the users code. After some discussion between ourselves, we decided the best way to handle this, would be to modify our existing language and implement these checks within the functions for each rule, as we will now discuss.

Firstly we needed a way to identify which lines may need to be stopped on by the debugger, so to do this we developed a function to read a file which contained the list of lines with breakpoints, and we then stored these lines in a `HashSet` to check back to later, the function is below:

```
public void readFile(){
    try{
```

```
        Scanner sc = new Scanner(new File("user-files/breakpoint
        try {
            String line = sc.nextLine();
            String[] arr = line.split(",");
            for (int i = 0; i < arr.length; i++){
                if (!arr[i].equals("")){
                    breakPointArr.add(i + 1);
                }
            }
            sc.close();
        }  catch(NoSuchElementException e){
            sc.close();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Now that we had our set of breakpoints, we were able to begin to implement our logical checks to our debug version of the language. Lets look at an example function for context of how this works:

```
@Override
public Object visitComment(startParser.CommentContext ctx){
    ArrayList<Integer> comments = new ArrayList<Integer>();
    int firstLineOfComment = ctx.start.getLine();
    comments.add(firstLineOfComment);
    // Add all lines covered by the comment
    Token startToken = ctx.start;
    Token stopToken = ctx.stop;
    int startLine = startToken.getLine();
    int stopLine = stopToken.getLine();
    for (int i = startLine + 1; i <= stopLine; i++) {
        comments.add(i);
    }
    // or check if anything in breakPointArr is in comments
```

```
    for (int i = 0; i < comments.size(); i++) {
        if (breakPointArr.contains(comments.get(i)) &&
        !linesStoppedOnSoFar.contains(comments.get(i))) {
            int line = comments.get(i);
            linesStoppedOnSoFar.add(line);
            breakpoint(line);
            //remove the line from breakPointArr
            breakPointArr.remove(comments.get(i));
        }
    }
    return null;
}
```

In the above function, we start by finding the first and last token of a comment rule, and then use these to find the first and last line the rule appears on (comments could span lines). Then we add all lines the comment appears on to a local array. Then for each line, we check if that line also appears in the breakpoint array. If it does, and we have not stopped there (this is important as we could have multiple rules on one line, and we would only want to stop once per line), we call the breakpoint function, and keep note of the line we stopped on. This similar logic is used throughout our visitor, in some places needing to be much more complex to solve the problem, such as loops and conditions, which we will discuss later on.

The breakpoint function simply calls a function in our `callDjango.java` file named `pauseCode`, which then updates the frontend to display the code being on a certain line, and then a function in our `fileChecker.java` file is called named `checkFile` which monitors an instruction file in user-files directory for each user, and when the user finally advances the breakpoint, the file text is changed from paused to continue, allowing the visitor function to continue executing the code passed to it. Here is the main loop from the `checkFile` function:

```
// Run an infinite loop to continuously check for changes
while (true) {
    WatchKey key = watchService.take();
```

```
    for (WatchEvent<?> event : key.pollEvents()) {
        // Check if the event is a modification event
        if (event.kind() == StandardWatchEventKinds.ENTRY_MODIF
            // Get the context of the event (file name)
            Path modifiedFilePath = (Path) event.context();

            // Check if the modified file is the one you're inte
            if (modifiedFilePath.equals(filePath.getFileName())
                // Perform your content checking logic here
                // Example: Read the file contents and perform c
                String fileContent = new String(Files.readAllBy
                if (fileContent.contains("continue")) {
                    key.reset();
                    return;
                }
            }
        }

        // Reset the key to receive further events
        key.reset();
    }
```

At the end of a program, once all lines have been visited and all breakpoints stopped on that need to be, the `visitProgram` function will call the `endCode` method from the `callDjango.java` file which indicates that the execution is now over in the backend.

Web IDE Frontend

The built-in web IDE provides users with a space to write, execute and save code.

We used Ace Editor as a framework to implement our built-in web IDE. Ace Editor is a third party open source embeddable code editor which is written in JavaScript. Given Ace's open source nature, we were able to alter some of its

source code in order to cater to the START language. For example we were able to write our own file to feed to the editor which provided rules for custom syntax highlighting. The Ace Editor also provided built-in JavaScript functions, such as functions for adding breakpoints and adding/removing code from the IDE to name a few. For example here is the code used to place a breakpoint on given line. An event listener listens for a click action within the boundaries of the IDE. Multiple checks are made; first ,if the debug mode is not dynamic (meaning the user should not be able to freely set breakpoints), a breakpoint is not set. It also checks to ensure the click action took place within the 'gutter' of the IDE. When all checks pass, a breakpoint is set in that 'gutter cell' using Ace's built-in function `setBreakpoint().` If a breakpoint is already present in that 'gutter cell', the `clearBreakpoint()` is used instead.

```
editor.on("guttermousedown", function(e) {
    var target = e.domEvent.target;

    if (debugMode != 2){
        return;
    }

    if (target.className.indexOf("ace_gutter-cell") == -1){
        return;
    }

    if (e.clientX > 25 + target.getBoundingClientRect().left){
        return;
    }

    var breakpoints = e.editor.session.getBreakpoints(row, 0);
    var row = e.getDocumentPosition().row;

    // If there's a breakpoint already defined, it should be rer
    if(typeof breakpoints[row] === typeof undefined){
        if (editor.session.getLine(row).trim() != ""){
            e.editor.session.setBreakpoint(row);
        }
```

```
    }else{
        e.editor.session.clearBreakpoint(row);
    }

    e.stop();
});
```

Writing code in the IDE operates identical to any regular IDE. Underneath the IDE there are various buttons which aid its operation. Each button sends a request to the backend which is where code execution takes palce.

- Run - Triggers code execution
- Continue - Used to resume code execution when code has paused due to reaching a breakpoint.
- Cancel - Used to cancel code execution at any time.

Measures have been taken to ensure that these buttons are not pressed in an unexpected order at any given time. Values are set in the session storage which are then checked when these buttons are pressed. For example when the ‘Run’ button is clicked, a session storage value is checked to determine if code execution is already taking place. If it is, a message is displayed to the user, and a second code execution is not triggered.

```
let running = sessionStorage.getItem("running");
    if (running == "true"){
        message("warn", "Code is currently running. Please wait
        return;
    }
```

The IDE supports various debug modes which the user can choose between, with an on-screen three-way toggle.

- Normal - Code will run from start to finish without pausing.
- Line-By-Line - Code will run, pausing after executing each line.
- Dynamic - Code will run, pausing after executing each line with a breakpoint. The user can place breakpoints on lines of their choosing.

When a user switches to normal mode, all existing breakpoints are cleared and the user is not able to place breakpoints. When a user switches to line-by-line, breakpoints are placed on every line and the user is not able to alter this. When a user switches to dynamic all existing breakpoints are cleared and the user is freely able to add/remove breakpoints as they please. All mentioned operations are handled using Ace's built-in functions.

When code is executed in the backend, the result is sent to the frontend via WebSockets (more on this in the 'Backend' section), and displayed in the 'Result' IDE. This IDE is set to 'read-only' mode. This is achieved when setting up an instance of an Ace Editor, as shown here:

```
var result = ace.edit("result", {
    theme: "ace/theme/tomorrow_night_eighties",
    mode: "ace/mode/text",
    minLines: 10,
    maxLines: 10,
    wrap: false,
    autoScrollEditorIntoView: true,
    readOnly: true
});
```

When setting up an instance of an Ace Editor we can also set the theme, mode (syntax highlighting rules), minimum and maximum lines to be displayed, text wrap and more. Any variable values resulting from code execution are also displayed to

the user, in the ‘memory’ table. This information is also communicated to the frontend by the backend via WebSockets.

Further information about how code execution operates in the backend is explained in the following section.

Backend IDE Operations

Our backend was primary responsible for handling the execution of our START jar file and manipulating any output produced by the jar and sending it to the frontend to be displayed. I will now give a brief overview of some of the main backend endpoints responsible in executing our jar file.

The `upload_code` endpoint is responsible for initialising the execution of our jar file. It begins by taking information sent in the POST request and building multiple files that are either fed as arguments to the jar file or manipulated by the jar at some point during execution. Two files are created in the user-files directory with the following titles, followed by a unique uuid for each user in order to distinguish a specific user’s files from others:

- input - A text file containing the source START code taken from the web IDE to be executed by our jar.
- breakpoints - A text file containing information about which lines in the IDE have been highlighted with a breakpoint.

The jar file is then initialised as a background subprocess. The choice to have the subprocess as a background action is to allow the execution to be cancelled if the user wishes to do so.

```
command = [java_path, '-jar', jar_path, file_path, token, u]

                try:
                    process = subprocess.Popen(command, stdout=s
```

```
                    return JsonResponse({'process': process.pid
```

A process ID is returned to the frontend to be stored in the session storage. This allows an optional subsequent cancel command to target the specific subprocess easily.

The `pause_code` endpoint if called by the jar file and is triggered when a breakpoint in the code is hit. The jar file sends the line number on which execution is paused. The backend then sends this number to the frontend via a WebSocket so that the line can be highlighted within the IDE for the user. At this point another file within the user-files directory is read; memory. The memory file is a csv file which contains keys and values of the START codes memory. This information is also sent via a WebSocket to the frontend so that it can be displayed to the user. Below is the code responsible for sending this information to its relevant WebSocket:

```
layer = get_channel_layer()
    async_to_sync(layer.group_send)('breakpoint', {'type': 'send
    async_to_sync(layer.group_send)('memory', {'type': 'send_mes
```

The `step_code` endpoint is called from the frontend when a user wishes to resume code execution. The endpoint writes to a file; instruct. This file is monitored by the jar continuously. When a change is made by the `step_code` endpoint the jar file continues execution. The endpoint will also update the breakpoint file. This allows any breakpoint adjustments made by the user to be registered by the jar file.

The `cancel_code` endpoints allows the jar file execution to be cancelled by the user at any time. The endpoint receives the aforementioned process ID from session storage. It then uses Python's psutil and os modules to kill the jar's subprocess.

```
pid = int(pid)
working_dir = os.getcwd()
try:
```

```
    if psutil.pid_exists(pid):
        process = psutil.Process(pid)
        if process.is_running():
            # Check if the process is a JAR execution
            cmdline = process.cmdline()
            if any("java" in arg for arg in cmdline) and any(".j
                # Send SIGTERM signal to kill the process
                os.kill(pid, 15)
```

The endpoint then deletes any remaining files from the user-files directory associated with the current jar execution, ensure a clean environment for the next execution. It is important to note that this endpoint is also called before a user switches pages within the frontend to ensure code execution happens only on pages were it is expected.

The `end_code` endpoint is called by the jar and is responsible for informing the server that jar execution has completed. It sends the memory through its WebSocket to be displayed one last time and also deletes any remaining files from the user-files directory associated with the current jar execution, ensure a clean environment for the next execution.

User Accounts & Password Reset

The user model, as well as user login, logout and registration functions, are all carried out by Django's built-in User model and functions, with minor adjustments. For example, the first name and last names in the model have been switched to unrequired.

```
class RegisterUserForm(UserCreationForm):
    email = forms.EmailField(widget=forms.EmailInput(attrs={'cla
    first_name = forms.CharField(max_length=50, required=False,
    last_name = forms.CharField(max_length=50, required=False, w


    class Meta:
```

```
        model = User
        fields = ('username', 'first_name', 'last_name', 'email


    def __init__(self, *args, **kwargs):
        super(RegisterUserForm, self).__init__(*args, **kwargs)

        self.fields['username'].widget.attrs['class'] = 'form-c
        self.fields['password1'].widget.attrs['class'] = 'form-
        self.fields['password2'].widget.attrs['class'] = 'form-
```

This form is used during user registration, and saves the user model to the database. Although the Django built-in functions are used to login/logout, we did extend the user functionality with other a password reset feature.

If a user has forgotten the password used to login to their own account, we included a feature where they could reset their password by sending a link to the email associated with the account.

We get the username from the user, and then retrieve the email associated with that username, we then generate a link that contains that users token and uuid, so the link is unique to each user, as seen below:

```
def send_reset_email(request):
    EnterUsername = request.POST.get('username')
    print(EnterUsername)

    user = User.objects.filter(username=EnterUsername).first()
    print(user.password)
    print(user.email)

    if user:
        token = default_token_generator.make_token(user)
        uid = urlsafe_base64_encode(force_bytes(user.pk))
        subject = 'Password Reset Request'
        message = f'Hi {user.username},\n\nPlease click the link
        password:\n\nhttp://127.0.0.1:8000/reset-password/?uid=
```

```
        \n\nThanks!'
        send_reset_email_to_user(subject, message, user.email)
    return redirect('email-sent')
```

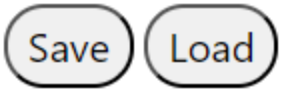
Once the user has navigated through the steps to reset the password, and have entered the new password, the following function is called:

```
def update_password(request):
    uid = request.POST.get('uid')
    token = request.POST.get('token')
    new_password = request.POST.get('password2')

    try:
        uid = urlsafe_base64_decode(uid).decode()
        user = get_user_model().objects.get(pk=uid)
    except:
        user = None

    if user is not None and default_token_generator.check_token(
        user.set_password(new_password)
        user.save()

    return redirect('login')
```

We use the built in set_password() with the new password to update that users password in the Django database.

Code Saving and Loading

When a user is logged in the ‘save’ and ‘load buttons will be visible to them below the the IDE.



These buttons only being displayed if a user is logged in is achieved by using Django's built-in HTML 'tags'. The HTML for the buttons is wrapped in an 'if' condition:

`{% if user.is_authenticated %}`

If the condition passed the HTML within the tags will be rendered on the page.

When a user clicks 'save', they are greeted with a pop-up modal in order to confirm their save. After naming and confirming the save, a backend POST request is made to save the code session to the database. For our database we use SQLite which has integrated functionality with Django, which allows for quick and efficient database manipulation. The session is saved to the Session table in the backend view `save_session()`. A new Session instance is created, containing the session content, its associated user and its title.

```
title = request.POST.get('title', '')
            session = Session(username=request.user.username, se
            session.save()
            return JsonResponse({'result': 'success'})
```

The session model is defined with the `model.py` file in the Django project.

```
class Session(models.Model):
    username = models.CharField(max_length=150)
    created = models.DateTimeField(default=timezone.now())
    modified = models.DateTimeField(auto_now=True)
    title = models.CharField(max_length=50, default="Untitled")
    session = models.TextField()

    def __str__(self):
        return self.username
```

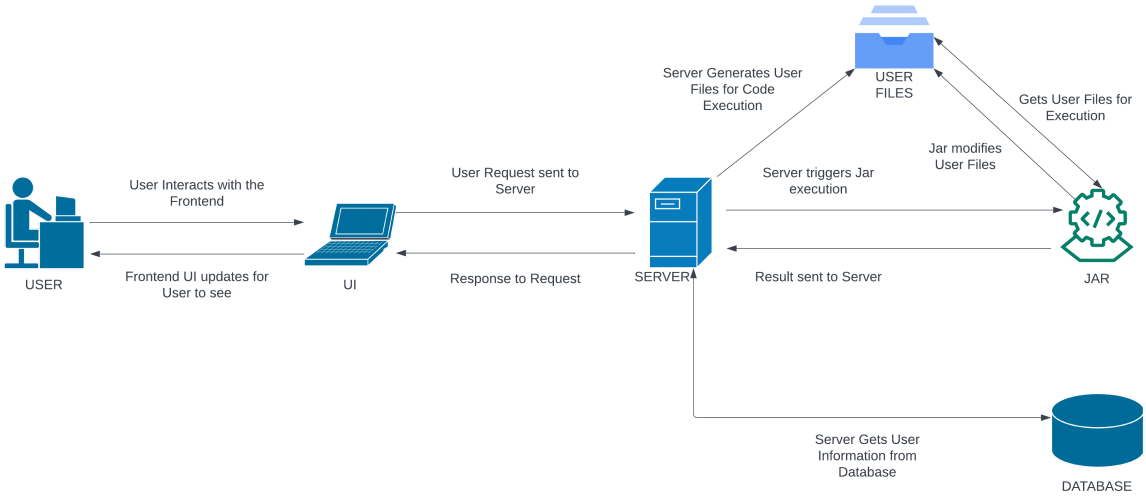
Loading a session works in reverse. A backend GET request is made, and sessions ae retrieved from the database, filtered by the user's username. These sessions are then returned to the frontend for the user to select.

```
@login_required
def get_sessions(request):
    if request.method == 'GET':
        user = request.user
        sessions = Session.objects.filter(username=user.username

        if len(sessions) == 0:
            return JsonResponse({'session': []})

        sessionsData = []
        for session in sessions:
            sessionsData.append({
                'title': session.title,
                'created': session.created,
                'modified': session.modified,
                'session': session.session
            })

        return JsonResponse({'session': sessionsData})
```

Learning Resources

(will do when fully done)

System Architecture Diagram

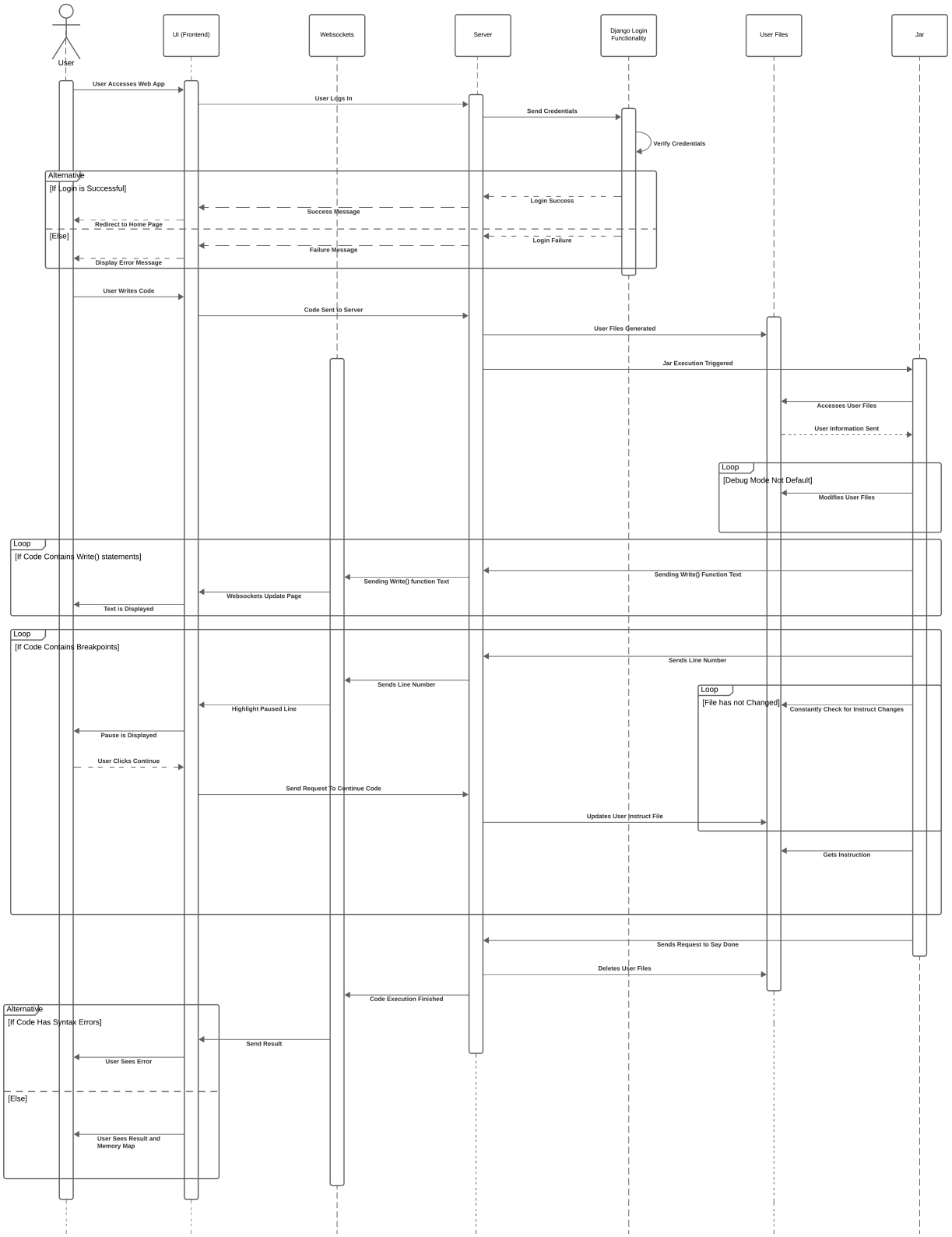


Above is the System Architecture Diagram for the START Web Application. As we can see, it begins with the User interacting with the UI of the Web Application, perhaps to login, or to write code, any interaction by the user is seen on the frontend UI. From here whatever task the User is trying to carry out will send a request to the Server. From here we have some options of what is happening:

1. The User is accessing account information from the Database, perhaps trying to login, or reset a password. If this is the case the Server will get the information from the Database and then respond to the User request, and the user will see the updated UI based on what task they are completing.
2. The User is executing some code, so the Server generates the User Files for execution, and then triggers the Jar execution with the User's code. From here the Jar will use the User Files to complete the execution and generate a result to be sent back to the Server. Again the Server will respond to the request to run code, and the User will see the updated UI.

High Level Design

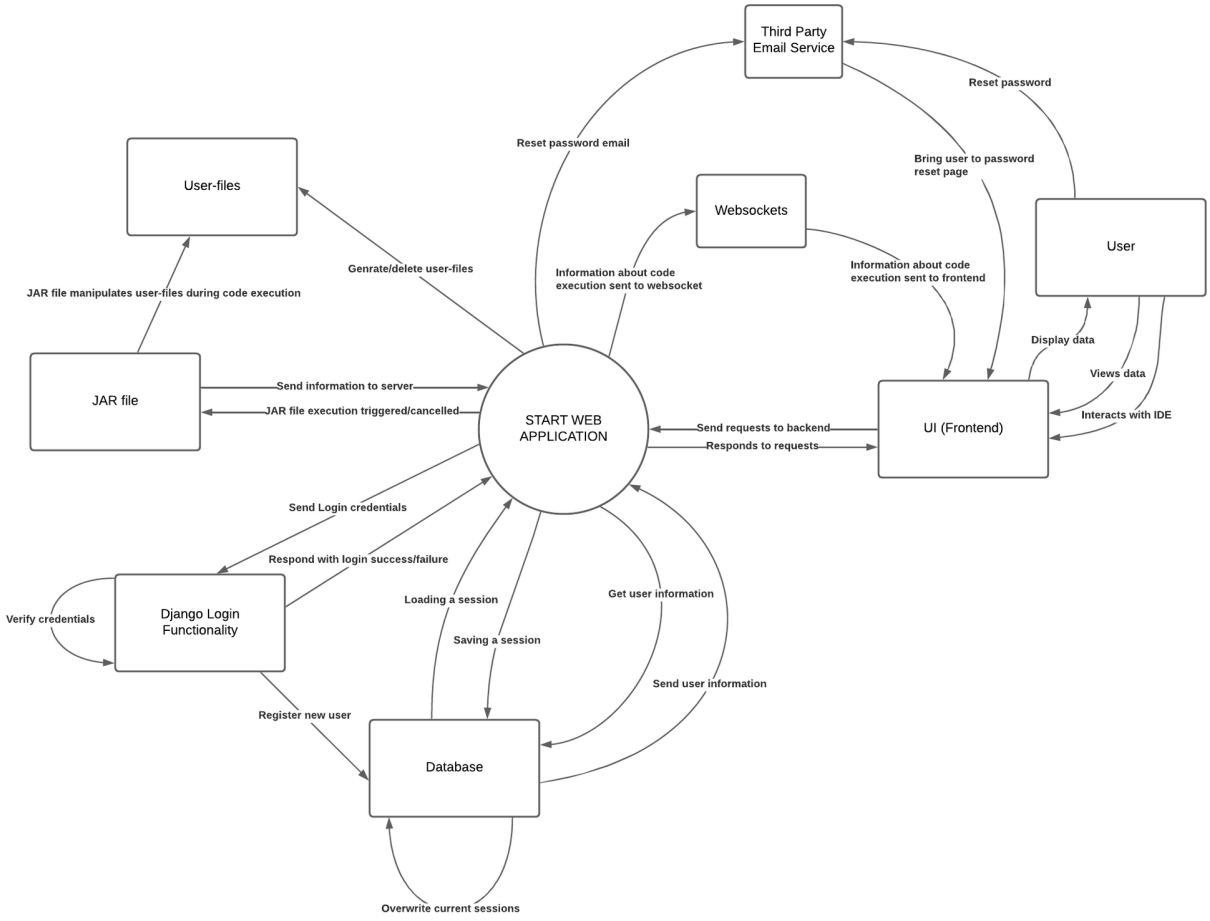
Sequence Diagram


User
UI (Frontend)
Websockets
Server
Django Login Functionality
User Files
Jar
User Accesses Web App
User Logs In
Send Credentials
Verify Credentials
Alternative
[If Login is Successful]
Login Success
Success Message
Redirect to Home Page
[Else]
Login Failure
Failure Message
Display Error Message
User Writes Code
Code Sent to Server
User Files Generated
Jar Execution Triggered
Accesses User Files
User Information Sent
Loop
[Debug Mode Not Default]
Modifies User Files
Loop
[If Code Contains Write() statements]
Sending Write() function Text
Sending Write() Function Text
Websockets Update Page
Text is Displayed
Loop
[If Code Contains Breakpoints]
Sends Line Number
Sends Line Number
Loop
[File has not Changed]
Constantly Check for Instruct Changes
Highlight Paused Line
Pause is Displayed
User Clicks Continue
Send Request To Continue Code
Updates User Instruct File
Gets Instruction
Sends Request to Say Done
Deletes User Files
Code Execution Finished
Alternative
[If Code Has Syntax Errors]
Send Result
User Sees Error
[Else]
User Sees Result and Memory Map

Above is the sequence diagram for “User runs some code”. It starts with the User accessing the Web App and then logging into the app, which requests the server to log them in. The Django Login Functionality then attempts to verify the login, if successful the User is redirected to the Home Page, otherwise an error is seen. From there the user writes some code that is sent to the server, which then generates the User Files for execution, and the Jar is then triggered, which access the User files. From here we have some optional conditions that may happen. If the debug mode is not set to default the User Files will be modified. If the program encounters a write statement, the text is sent to the Websockets which update the frontend UI for the User to see. If the Jar encounters a breakpoint line, the line number is sent to the websockets, which highlights the correct line for the User to see. And then when they click continue to advance the code, a request is sent to continue the code through the Server, which updates the instruction in User Files. The Jar then gets its instruction from this file. This continues until no breakpoints are left. Once the Jar finishes executing, it tells the Server which deletes the User Files. The Server then tells the Websockets to send the result to the frontend UI, and if an error is generated from the code, it is displayed, else the result, (if any) is shown to the User.

Below is the accompanying Context Diagram for this Sequence Diagram, which shows the systems flow in its entirety. As shown it includes any 3rd party email service that is used by the User to set up their account. In the case that they need to reset their password, the email is sent to this account, which when the link is clicked, brings the user to the reset page in the frontend UI.

Context Diagram



Problems Faced and Solutions

Sending Output to IDE

One of the main technical problems we faced during the development of this project was sending output generated from executing the Start jar file. Our original implementation of the jar file from our 3rd year project included printing any output to standard output (terminal). We initially used this as a .... However, during development we noticed that this output would only display in the IDE upon code execution completion which was not ideal in the scenario any print statements were present prior to a breakpoint, in which case no output would be displayed when the breakpoint was triggered, which is not how a typical debugger operates. We also needed to alter the jar execution to be an asynchronous operation to facilitate code cancellation.

To overcome this issue we decided to implement WebSockets as a way to send output to the frontend of our system as it is required, and not all at once. Within our jar file we wrote a method which would an endpoint on our Django server, sending any relevant information to the backend. Here is an example of a call we used in order to send any data from a print statement to the backend:

```
public static void printLine(String line, int currentLineNum, in
        try {
            // Set the URL of your Django server endpoint
            String url = "http://localhost:8000/print-line/";

            // Create the HttpURLConnection
            HttpURLConnection connection = (HttpURLConnection) u

            // Set the request method to POST
            connection.setRequestMethod("POST");

            // Enable input/output streams
            connection.setDoOutput(true);

            // Send the request
            // put the token in the header
            connection.setRequestProperty("X-CSRFToken", token);
            // set cookies value to the token value
            connection.setRequestProperty("Cookie", "csrftoken="
            OutputStream os = connection.getOutputStream();
            String output = line + " " + currentLineNum + " " +
            System.out.println(output);
            byte[] bArr = String.valueOf(output).getBytes();
            os.write(bArr);
            os.flush();
            os.close();

            // Check the response code
```

```
            int responseCode = connection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_OK) {
            } else {
                System.out.println("API request failed with resp
                System.exit(0);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

This information would then be received in our backend and sent to its relevant WebSocket:

```
layer = get_channel_layer()
    async_to_sync(layer.group_send)(
        'print',
        {
            'type': 'send_message',
            'line': line,
            'line_number': line_number,
            'column': column,
            'id': id
        }
    )
```

A Python WebSocket consumer would then receive this information from the channel and be able to be displayed in the IDE using JavaScript.

We in turn used WebSockets to send other information from the jar to the backend, such as notifying the server of when the code execution was paused or when it had completed.

Removing auto-generated output from browser console

Originally the output auto-generated by ANTLR was being sent with the payload to the console of the browser, and this was filling the browser cache up with lots of useless information about each program being ran. To fix this, we needed to silence the output stream coming from the driver file running the visitor. The following code suppresses the output from being generated:

```
PrintStream originalOut = System.out;
System.setOut(new PrintStream(new OutputStream() {
    public void write(int b) {
        // Do nothing to suppress output
    }
}));
```

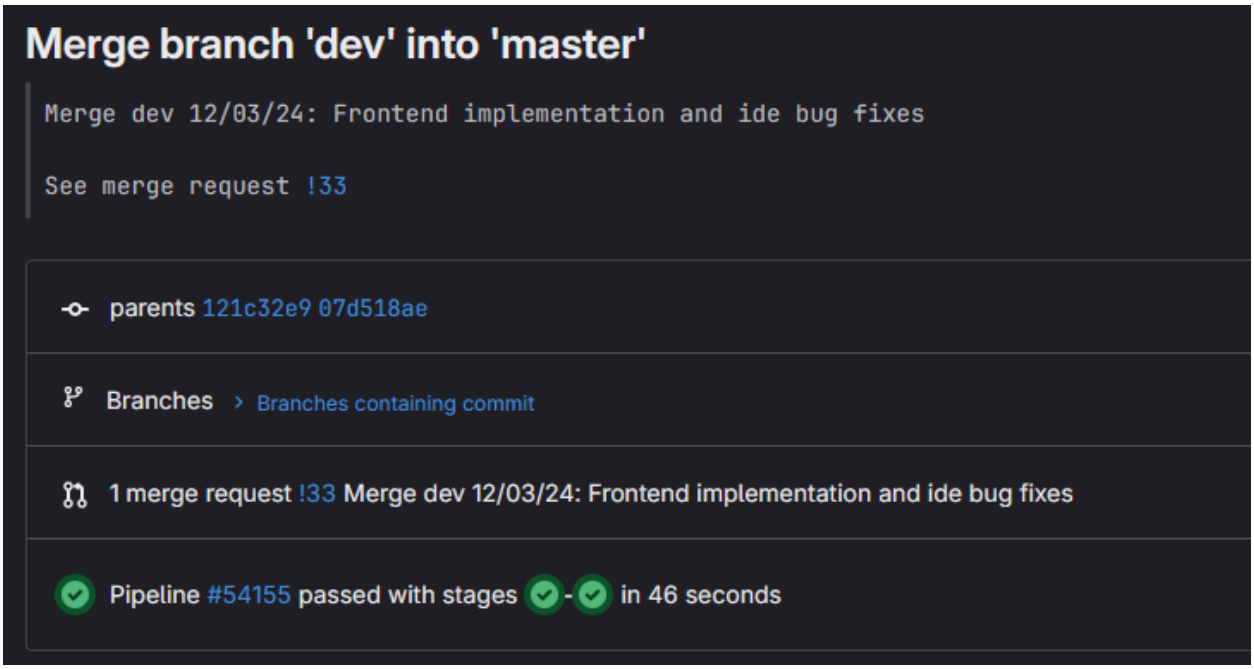
At the end of the driver file, we then reset the output stream to be the saved `originalOut`:

```
System.setOut(originalOut);
```

Making GitLab Runner

When setting up our CI/CD for our project, we ran into an issue with our docker build not happening on Gitlab. At first we assumed we had implemented our Dockerfile incorrectly, but after research and investigation we could not find any issue. We decided to contact the head of Gitlab in DCU Dr. Stephen Blott who informed us that for security reasons, DCU Gitlab would not allow Docker images or containers to run with the Gitlab runner, however we could create our own runner with our own machines being used as the server to run the tests.

With this in mind we researched how indeed to do this, and came across the GitLab Runner download for Windows. We installed this onto our own machines and linked it with our project repo, which when we pushed to a branched with the runner enabled it would run our tests locally on our machine and send the results to our repo on Gitlab.



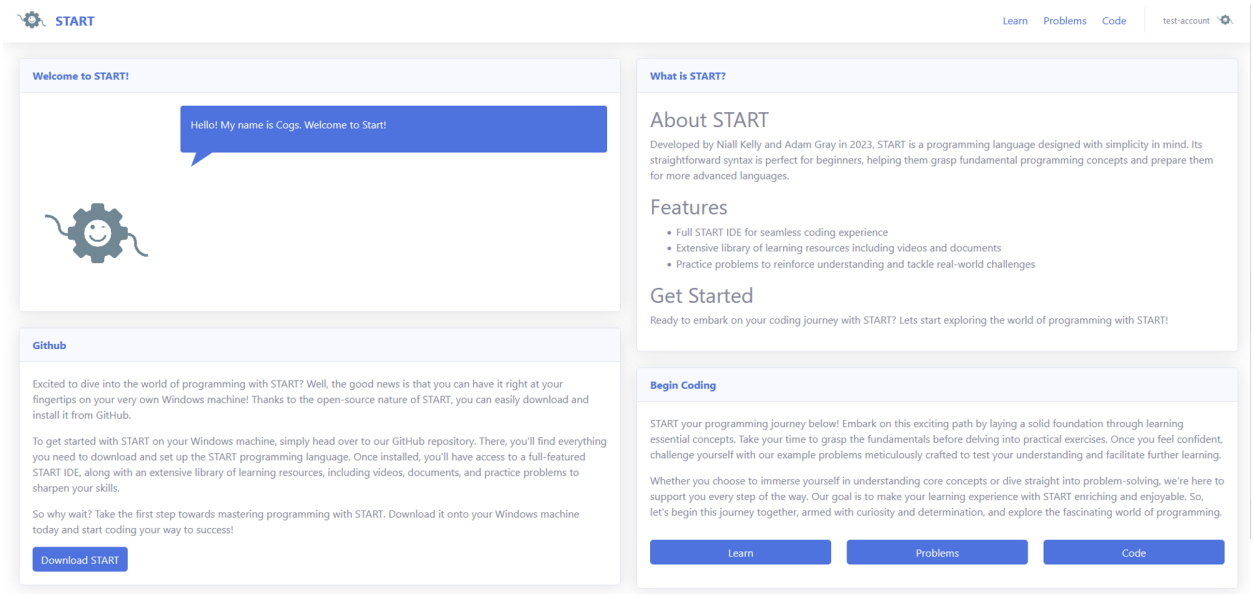
Above is an example of our pipeline passing our tests with our own runner.

Final Design and Implementation

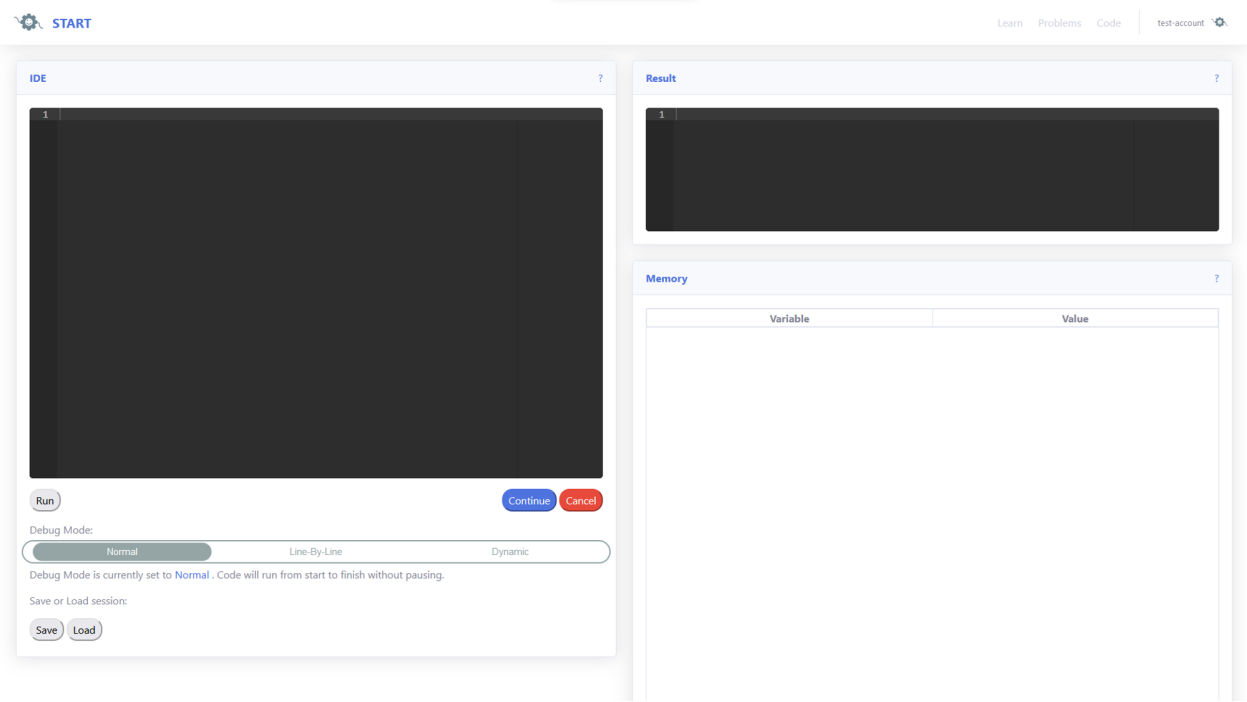
Pages of the Web App

Below are a few of the final implementations of the main pages of the web app:

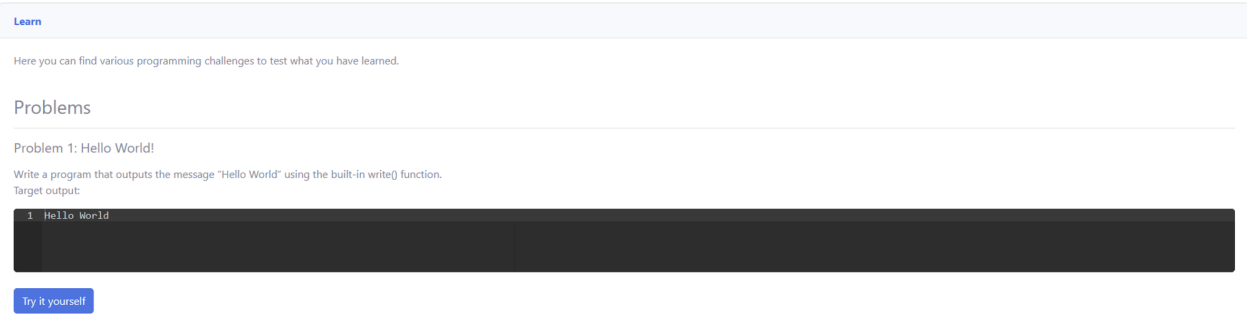
Home Page

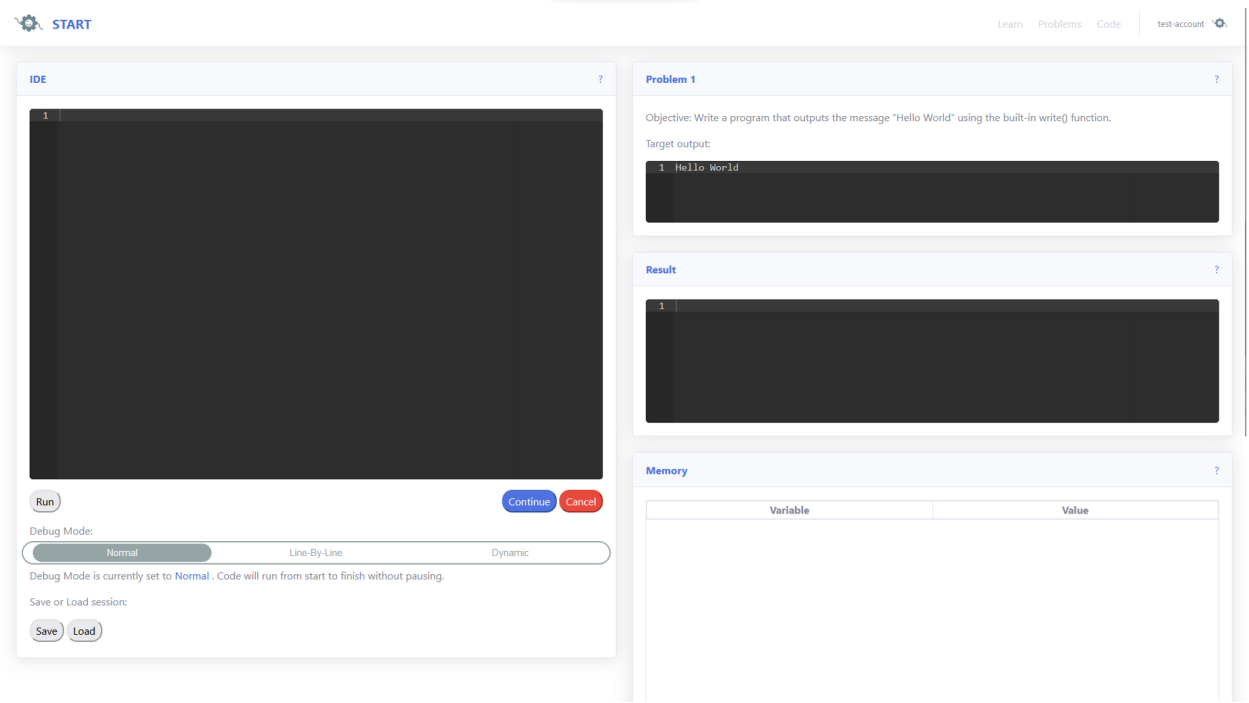


IDE Page



Problems Page





UI Design & Evaluation

When it came to designing our UI originally, we decided to follow the 10 rules laid out in Jakob Nielson's book Usability Engineering. Below are these rules, followed by an evaluation of how well our web app followed these initially chosen rules:

1. **Visibility of system status.** Users should always know where they are and what's going on.
    - Following this rule, every page has titles of the features on the page, helping the User to recognise instantly what page they are on. We believe we have followed this rule perfectly and without exception.
2. **Real world - system match.** The system should mirror the real world of the user as much as possible. Use language, concepts, etc. that are familiar to the user. Order the processes/screens in a way that is meaningful and logical to the user.
    - This rule was one that did not fully apply to our application, however, where it did was the language we used in the web app. With this in mind, we made sure to keep all language used simple and easy for someone knew to programming to understand.

3. **Control and freedom.** Don't "trap" the user. Support clearly marked exit, undo, and redo functions. Don't force them into a long linear sequence of operations with no escape.
    - We ensured every page had a way of navigating back to where the user came from or to move to a different page. The User is never trapped or forced to only move forward through pages in our web app.
4. **Consistency and standards.** Use objects and phrases consistently. Follow platform conventions. Here is a checklist of specific items to watch for.
    - We kept the colour scheme and font the same all across our pages, allowing for visual consistency for the Users.
5. **Recognition not recall.** Provide visual objects, actions, and options (e.g. cue cards) to assist the user for navigation and input activities. Don't expect they will memorize commands.
    - We tried our best to follow this rule by making all clickable links or new information stand out against its background, making it easily visible for the User.
6. **Flexibility and efficiency of use.** Accelerators (unseen by novice users) can speed up interaction for expert users. Allow users to customize frequent actions whenever possible.
    - This UI principle is not present in our design.
7. **Aesthetic and minimalist design.** Visibility of rarely needed information should be avoided. The more information that appears on the screen, the less visible each unit of information becomes.
    - We kept the amount of content on each page to an absolute minimum, and there is no rarely needed information. All information present is needed. We have little to no visual clutter in our design.
8. **Online help and additional documentation.** Though a well designed system can be used without documentation and help, supplemental information may still be necessary. Keep this information tied to user tasks, support easy to use search functions, and don't make this section too large.

    - Documentation on how to use our language is easily available through learning documents and videos.
9. **Effective error handling.** Assist users to recognize, diagnose, and recover from errors. Don't just tell them there's an error, suggest corrective action whenever possible.
    - Any error a user may make has an appropriate error message displayed to the user, be this with the coding problem they are trying to solve, or for example trying to log in with the wrong password.
10. **Error prevention.** A design that prevents errors from occurring is better than a good error message.
    - While not directly present in our design, our minimal design makes it tougher for Users to make obvious errors.

Looking at our design, our UI follows 8 of these rules absolutely, as well as partially following another. Scoring our overall rules followed out of 10, that gives us a score of 8.5/10. which we are extremely happy with in our final design, given it has been many months since we originally discussed trying to follow these rules are closely as possible.

Future Work

Given that this project was spawned from our passion to continue our work on our START programming language, some future plans we have for the project include:

- Collecting more User feedback to allow us to make quality of life improvements.
- Host our web app and deploy it for public use.
- Market our Web Application to Users who we feel are target User’s of our application.
- Consider revisiting START’s core features to see if we feel any new features could be developed and added to the language, however they would need to provide more ways of learning for a User, without overwhelming them.