

Group # 8

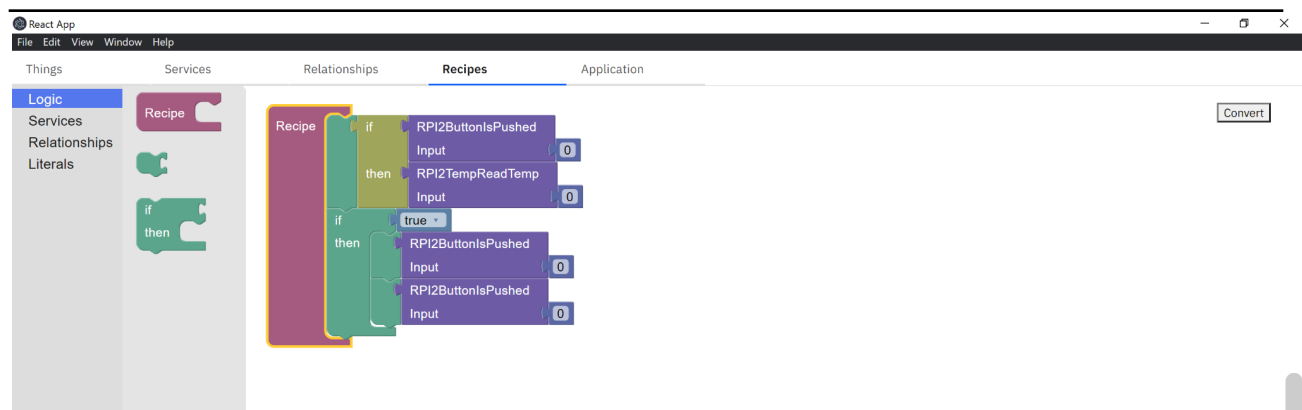
An IDE for Atlas Thing Architecture

Li, Jiahao

Li, Sijia

Liu, Xuanting

Zhu, Qiyue



1. Introduction (about 1 pages)

Passive service query and utilization for IoT services within the designated virtual smart space with minimized effort has been the challenge the class attempts to tackle throughout the semester. While Atlas IoT framework along with the Atlas IoT DDL provides the basic functionality and connectivity to advertise and utilize individual IoT microservices, none of them alone provides a structural means to subscribe, manage, and utilize IoT microservices and their interaction/relationship to efficiently develop applications on a larger scale. In particular, with the IoT DDL providing the elementary building blocks for an IoT application by masking currently available IoT microservices and signifying the relationships among currently bounded IoT services, the challenge arises when clients attempt to interpret these available IoT resources in meaningful ways. Additionally, clients will also have difficulty to readily utilize the IoT microservices or relationships in application composition without knowing the running conditions and boundary of the services. Finally, a clear, interactive graphical user interface is also needed for the client, giving the client the perspectives of the IoT development resources available and the logical role each IoT services or relationship fulfills. Therefore, the aim of this project, Atlas IoT IDE, is to create a client side Atlas IoT application development/management environment where

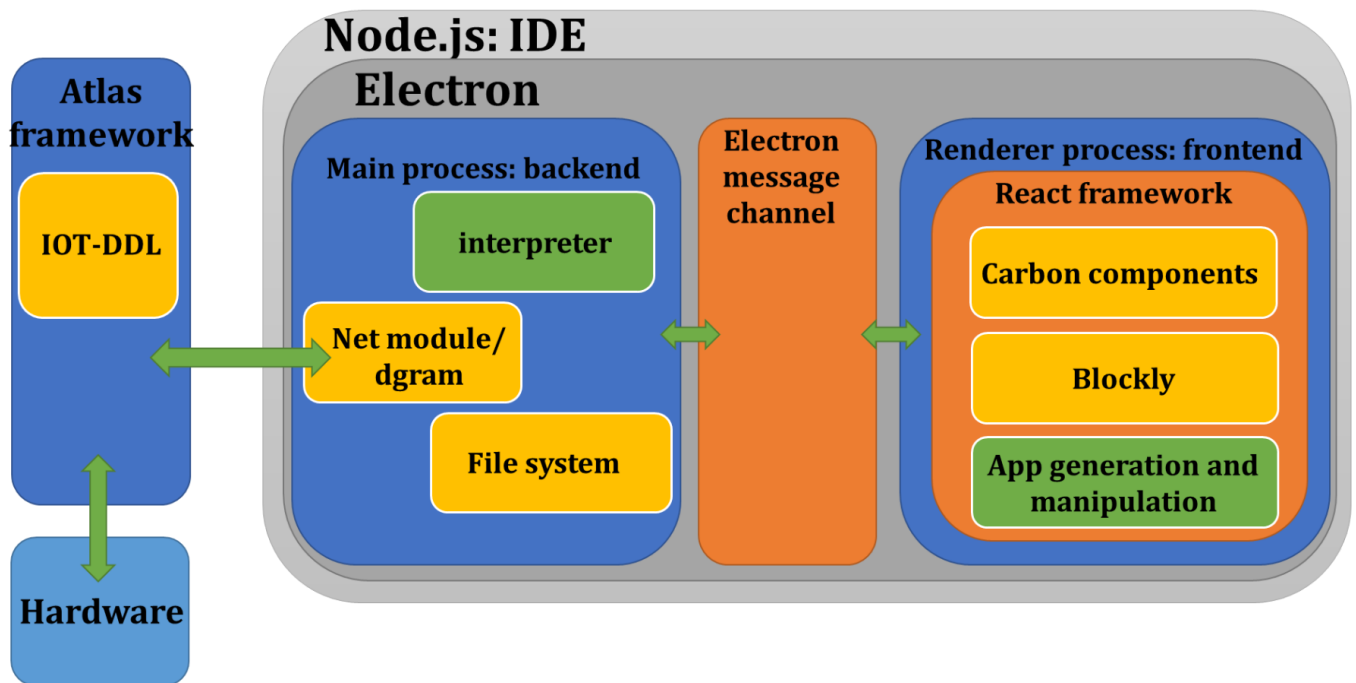
the client will be able to receive advertised IoT things, microservices and/or their relationship within the current virtual smart space as the resource for IoT application development, create IoT application with the given IoT resources, save and load Atlas IoT applications. While missing some features in specific relationship evaluation due to the hardware limitation of the current IoT microservices used by the development group, the final product being presented at the current development stages fulfills the basic functionality required by the project specification including evaluating IoT application syntax tree correctly, composing IoT application graphically, load or save IoT application in text file, and executing composed or loaded IoT applications. Particularly, providing that IoT microservices and relationships are correctly compiled and broadcasted by Atlas IoT DDL framework, the IDE can act as the service broker and app builder, integrating IoT services and relationships masked by IoT Atlas DDL into potentially meaningful applications while displaying the available IoT resource and currently available IoT applications in representative manner using diagram and graphical representation.

The following documentation will detailly describe the project and the project development process. Particularly, the documentation will provide perspective in the overall project design, highlight technical and design challenges faced by the development group during development phase, detail the implementation of the project, enumerating future development or maintenance plan, and accounting the group distribution among the developers.

Provide a link to your YouTube Video here in the introduction.

<https://youtu.be/CZZ6gmc96ds>

2. Project Design (about 1.5 pages)



Our goal is to develop a standalone IDE for the Atlas framework, and it should work on Windows/MAC PC environments.

We use Node.js to be our basic and overall framework since we are all familiar with javascript and its framework. It has many useful open source modules to support our development. Generally, we use Electron to build the system. Specifically, we are going to use net, dgram, and file system to support our backend development and react with carbon components and blockly to support our frontend development.

The Electron is a useful node.js framework based on javascript and web technology to develop desktop applications. It is conceptually a browser for web pages so we can use a web tech stack like react to develop GUI. We are going to implement backend logic in its main process and frontend logic in its renderer process which is actually a controllable browser process.

To communicate between frontend and backend, we have many choices. One of them is to use a traditional RESTful API based on TCP/IP connection. But in electron, a mechanism called “Message Channel” is supported to communicate between different processes in the electron framework. We are going to use this message channel to communicate between the main process

and the browser process. Since this channel is integrated into the electron framework, it is more light-weighted and has better performance. We designed a few data formats for different channels (API) like “tweetMessage”, “runApp”, “stopApp”, “saveApp”, “loadApp”, “getApp”, and “syncAPP”.

Inside the frontend process, we will use react to be the overall frontend framework. It is stable and easy to learn and use. To draw UI easily, we use a UI library called “Carbon components” for react to boost our development. It provides tags, buttons, tables, and many other ready-to-use components. To realize “drag and drop” functionality, we use the google “Blockly” library and modify it to satisfy our needs. We will have 5 tabs, the first 3 tabs are to show the information received from tweets, the fourth tab (aka Recipe) is used for developing a new Atlas application using drag & drop, the fifth tag is for managing (run, stop, save, load, and modify) all the applications in the memory.

Inside the backend process, we need to do the logic calculation as an interpreter for the Atlas application, and also do the communication job between frontend, backend, and Atlas frameworks on the Raspberry Pis. For the interpreter, we will write our own data structure and algorithm to estimate, execute and call every statement. We will make sure the user has the ability to stop the application between two statements. We will use net module to send API calls to the Atlas framework and we will use dgram module to receive tweets from the Atlas framework and synchronize these information with frontend. And of course, we will use file system in node.js with the help of the operating system to save and load applications and extra information along with it.

As for the IOT-DDL on the Atlas framework, we will use our old description for Lab4 and do some necessary modifications to make it possible to test if/then statements and relationships. The way how we connect the hardware is almost the same with Lab4.

The system is intended to use with connection to a smart space, IP broadcast address enabled. Before starting, the user should provide his/her own IP address. The application should run on a Windows/MAC environment, with one terminal executing “npm start” and another terminal executing “npm run electron-dev”. And then a desktop application will show up with all the functionality needed for an Atlas IDE.

2.1 Challenges Faced (about 1 page)

Accounting the technical and design challenges faced by the development group, several backend design and technical challenges were encountered by the backend development team.

At the very first beginning, the very first challenge we faced was how to receive multicast tweets using nodejs. Since the things running Atlas framework can receive tweets from each other, it seems not a problem with the router. However, even though we tried to run the code that attempts to receive in NodeJS, we weren't able to receive. Finally we find out that on Windows, the host's ip address needs to be provided while calling 'bind' and 'addMembership', while on macOS, the host's ip address must not be provided.

Beginning with the IDE backend service design, the first primary challenge faced by the backend developers is the communication mechanism between the frontend component/window and the backend services in electron framework. Different from the traditional web development strategy where the interaction between frontend client and backend services is achieved through RESTful api, electron provides its own communication pipelines between browser window process, the chromium sandbox containing the frontend client interfaces, and the main process, the main running process where the backend services are utilized. While the pipeline uses two interprocess communication module, ipcMain and ipcRender to achieve main-window communication through callback on specific channel/event being passed into the communication module and invoked by main or window process, the issue, besides mastering the communication mechanism of the pipeline itself, is to integrate the ipcRender module to the existing React component. It took some significant time of research and trials and errors to add the correct configuration to achieve the communication between react component and electron main process.

Additionally, when it comes to receiving tweets advertising the microservices or relationships from the IoT devices, there is also an issue with correctly receiving and parsing the tweets. As some of the special characters such as quotation markers are not properly casted by the Atlas IoT framework, lead to issue with parsing tweet string into JSON object and further obtaining service information from the tweets. The solution, without getting into the compiling mechanism of the Atlas IoT DDL framework, is to adjust the string to JSON rule at parsing phase to cast or weed out special characters with issue.

When it comes to the frontend part, we found a library named "Blockly" developed by Google that quite suits the scenario here. However it's guide is so deficient that it took us about 5 hours to just








make an example work. We needed to look up to the very detailed API documentation to figure out how to implement what we want. Moreover, this “Blockly” library has strange bugs while using it with tabs. If a user switches to another tab and switches back, it sometimes gives a broken layout. We tried to fix it with some tricks, but it still appears randomly.

Also, in order to achieve the load/save app function, we use the POSIX filesystem library for writing and saving applications on client side storage. However, failing to account for the asynchronous environment in which the IDE function is going to be utilized, the function calls used were synchronous thus leading to issues with locating the file location specified by the users.

3. Implementation (about 1.5 pages, plus as much as needed for the screen shots)

We have described the language, platforms, and frameworks we are using in the “Design” part of this report so we are not going to emphasize it repeatedly here anymore. Generally, it is Electron with React based on javascript.

3.1 Backend code folder explanation and implementation details

 public	backend finish test	4 hours ago
 src	test stop	4 hours ago
 .gitignore	 init	14 days ago
 README.md	added react	14 days ago
 package-lock.json	blocks	22 hours ago
 package.json	blocks	22 hours ago

This is our overall folder. All the code for the backend is in /public and almost all the code for the frontend is in /src.

..		
media	blocks	22 hours ago
ReceiveTweet.js	update tweet	5 hours ago
favicon.ico	added react	14 days ago
index.html	added react	14 days ago
intepreter.js	fix statement 4	4 hours ago
logo192.png	added react	14 days ago
logo512.png	added react	14 days ago
main.js	open	8 minutes ago
manifest.json	added react	14 days ago
robots.txt	added react	14 days ago
sendMessage.js	execute service	yesterday

This is our backend folder, mixed with some frontend static resources. Maybe it is not the best design but it works perfectly well. The major logics are in “ReceiveTweet.js”, “interpreter.js”, “main.js”, and “sendMessage.js”.

“main.js” is like the main function for the whole project. All the message channel’s data structure design and basic implementation are in “main.js”. There are a few responsibilities:

- Define the data structure we are going to use in all message channels.
- Use ipcMain in electron to provide all the information needed by the frontend, including providing all the up-to-date tweet info, providing the ability to run an app, stop an app, get the info of all the apps, save an app or load an app
- Receive broadcast tweets and maintain storage for tweet info, apps info.
- Create the frontend browser window with proper attributes
- Integration with the functionalities of other code.

“ReceiveTweet.js” is responsible for receiving tweets whenever there is a broadcast and updating the data structure in main.js with an algorithm according to the type of the tweet. We need to define the local IP address and port number in order to use “dgram” to receive multicast tweets. Then we can switch the type of the tweet and update the tweet info data structure.

“sendMessage.js” is responsible for sending API calls to the Atlas framework. It uses the “net” module to establish a TCP connection to Atlas Framework according to the IP and port received by tweets. It will send a service call message and wait for the reply from the Atlas framework.

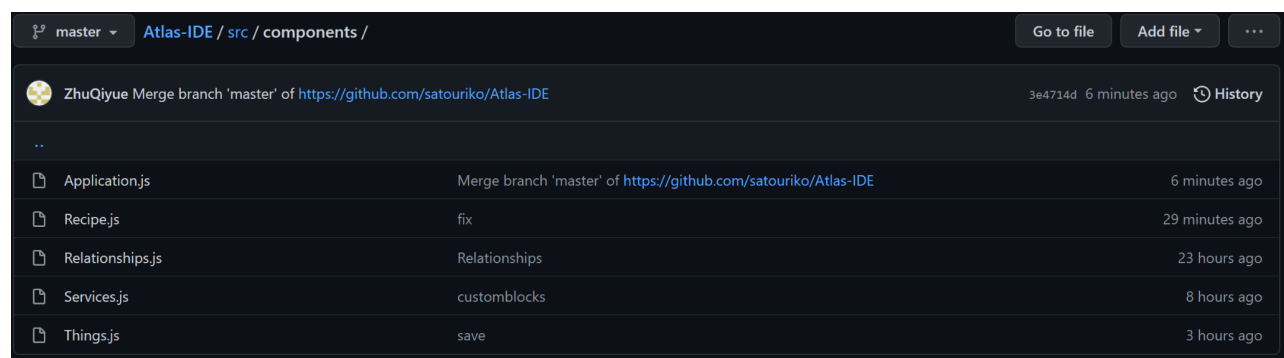
“interpreter.js” is the component to decide how to actually run a statement in an app. Each statement in our app has the following format:

```
let statement = {
  type: '', //service, relationship or ifthen
  thingID: '', //service if type is service, service1 if type is
relationship
  entityID: '',
  serviceName: '',
  serviceInput: [],
  relationshipType: '',
  thingID2: '',
  entityID2: '',
  serviceName2: '',
  serviceInput2: [],
  ifStatement: {},
  thenStatement: {}
}
```

Each statement can only be one of the three types: standalone service, standalone relationship, or an ifThen statement. Only ifThen allows nested statements. One statement doesn’t have to have attributes which are not for its type.

We wrote a function to execute a statement based on its type, called “executeStatement”. It is a very long function and all the services, relationships (together 4 kinds of relationships supported), and (nested) ifthen are supported. We have tested that they can be executed. To execute an app, we only need to execute it statement by statement and we check before we execute so we can stop any app in the middle. We also encapsulate a function for the service call here.

3.2 Frontend code folder explanation and implementation details



This is the components folder. In this folder, the code for each tab is returned. Things.js returns a table of the available Things as it is received from the backend by the multicast tweets. Services.js and Relationships.js serve a similar function, except that it also has a drop down menu to filter the

respective category by Thing Id. Recipe.js holds the Blockly workspace so that users can pick the Blockly block that suits their needs and build up their application as dictated by the recipe.

React App

File Edit View Window Help

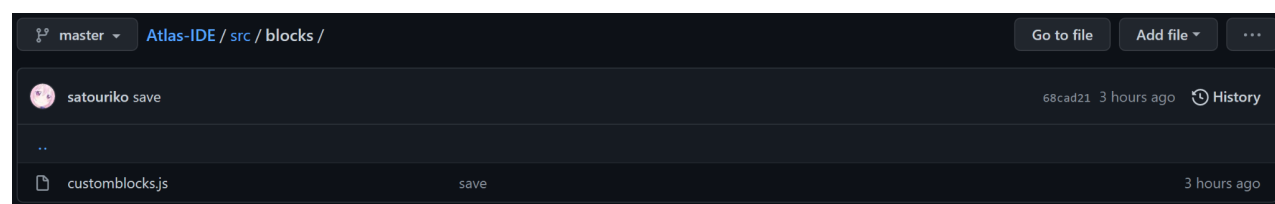
Things **Services** Relationships Recipes Application

Thing of Services

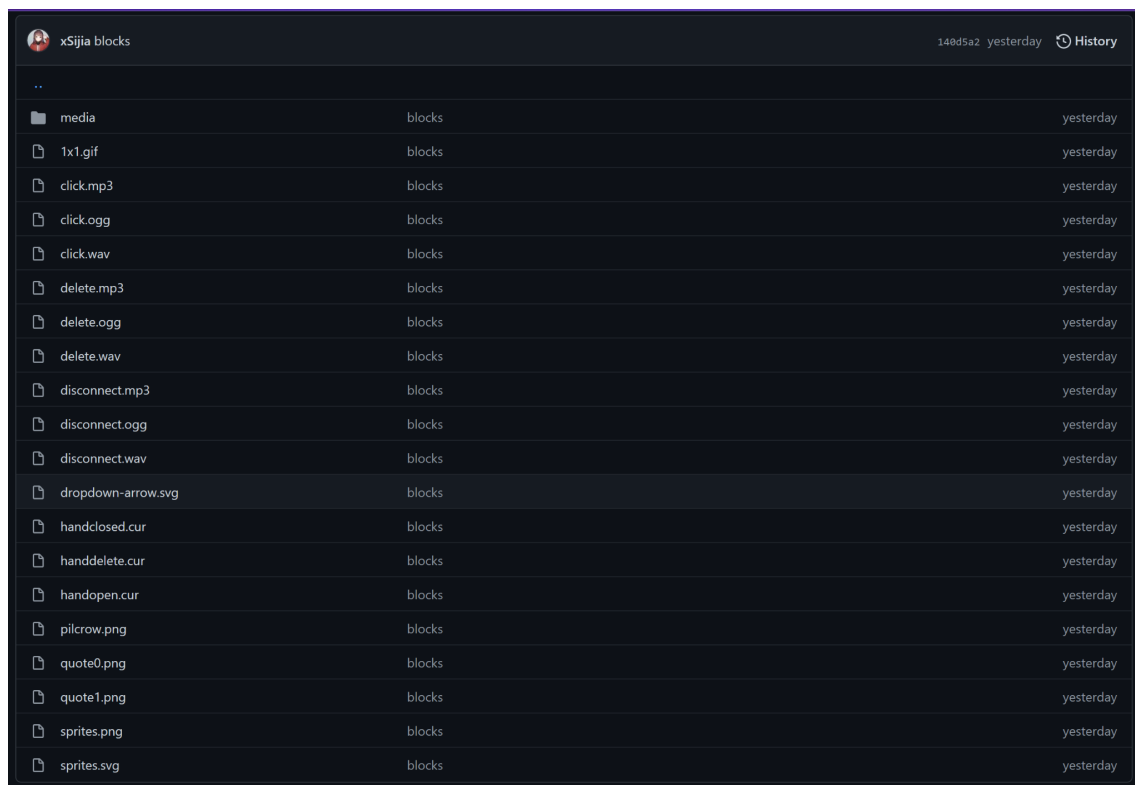
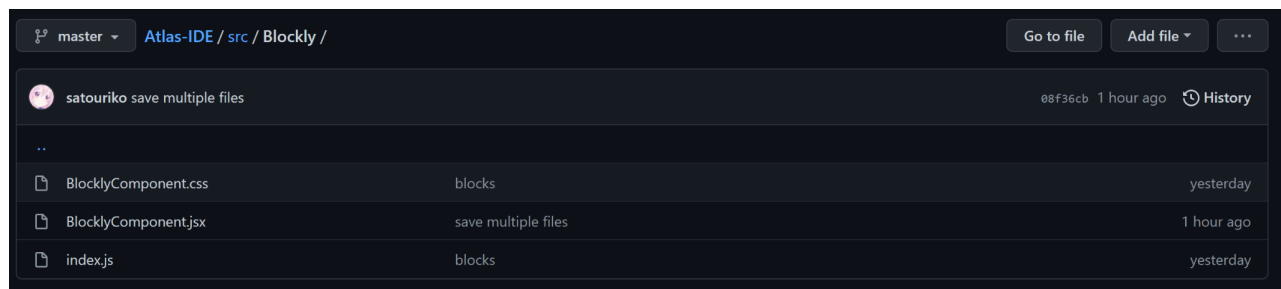
All

Thing ID	Entity ID	Name	Space ID	Type	Description	Keywords	AppCategory	Vendor
RPI1	BUZZER	BUZZ	LZL	Action	Ring the buzz with specific frequency and time		Time Alarms	
RPI2	Button	IsPushed	LZL	Report	Is the button pushed			
RPI1	LED	TurnOn	LZL	Action	Turn on the LED light		Lighting	
RPI1	LED	TurnOff	LZL	Action	Turn off the LED light		Lighting	
RPI2	Temp	ReadTemp	LZL	Report	Read the temperature.		Environment Monitor	

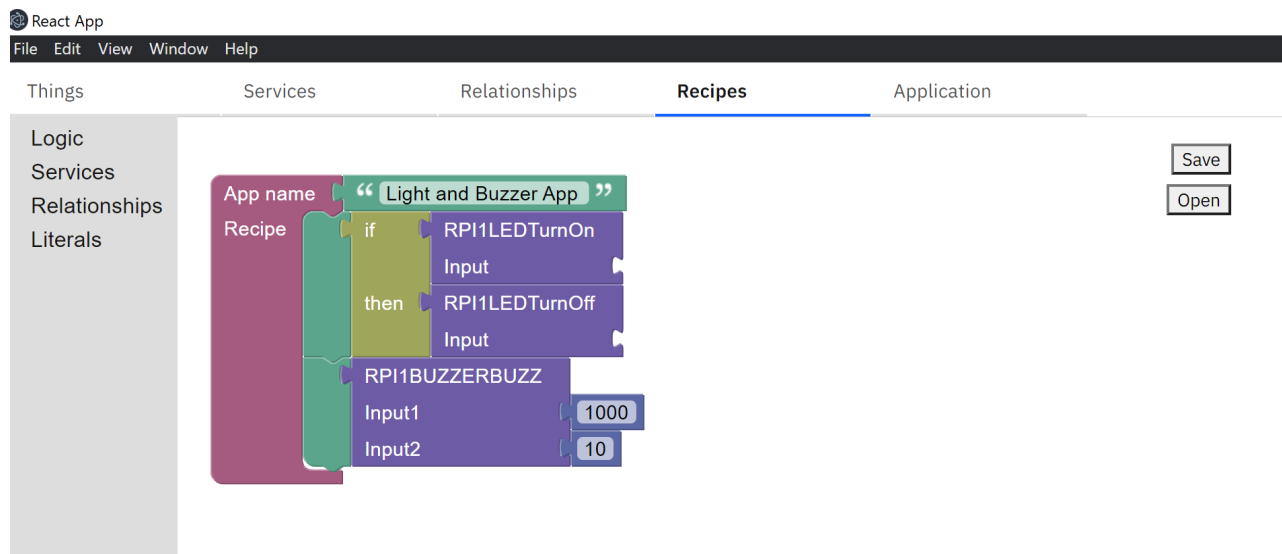
This is an example of the Services tab, with the Things and Relationship tab also set up similarly in a datatable view.



There is also a blocks folder which simply contains customblocks.js, where we define custom blocks built using Google's Blockly developer. A custom block can be a Service, or a Relationship. Here, we also defined what the custom blocks can connect to, and what type of code the connection of blocks generates.



Here are the Blockly component files from the library that was sourced from Blockly's react example, as well as other Blockly files that generate the Blockly connecting effects.



Example of building an application through the Recipes tab using Blockly. Once made, the application can be saved where it is downloaded as a .json file to the local drive, as well as stored in the cache to be accessed by the Application tap. Pre-existing .json application files can be opened from local storage and loaded into the ide, where it can then be saved and show up on the Application tab as well.

The application tab shows all the applications that have been loaded onto the IDE, in a table format, with each application having an option to start or stop the application, or even remove it from the IDE. There is also an option to import .json applications as well.

3.3 New things we learned and our experience

We are all new to the Electron framework but it is very easy to use and very powerful. Basically, all we need to learn and check is the official document of Electron <https://www.electronjs.org/docs/latest>. For the frontend part, it gives us no restriction. We can use React or any other frontend framework we would like to use since it is super flexible. With electron, there is no difference between writing a desktop application or a webpage. And in the backend, we can just think of main.js as the main function of the whole program, then we can write whatever logic we want in pure javascript.

The difficult part is to communicate between the frontend and backend in an efficient way, and we found “message channel” in electron after some survey. Once we learned how to use it, it is really easy to use. In the main process, we can use ipcMain to get access to any channel, and each channel is just like a RESTful API with its channel name. In the frontend, with some configuration

(not very obvious though), we can use ipcRenderer to get access to the message channel too. Then we can transform info between them and do whatever we want.

3.4 Screenshots and how to use our IDE

github address: <https://github.com/satouriko/Atlas-IDE>

1. Install node v16.13
2. `npm install`
3. Change line 5 of public/ReceiveTweet.js to HOST IP, also change line 4 and line 14 according to the multicast address and port the Atlas Framework is using. If you're running on macOS and could not receive tweets, try removing the second argument of line 14 and line 18.

Watch as available Things, Services, and Relationships start populating as it accepts tweets.

4. `npm start`, this starts the React dev server
5. After the dev server started, run `npm run electron-dev` that starts electron

> Create App recipes by conjoining blocks together

> Save App recipes locally

> See App recipes in Application tab and click on the start button to start the application, or the stop button to stop the application

> Delete or Load Application from Application tabs if you want to delete application from ide or add locally stored applications to ide, respectively.

React App

File Edit View Window Help

Things **Services** Relationships Recipes Application

Thing of Services

All

Thing ID	Entity ID	Name	Space ID	Type	Description	Keywords	AppCategory	Vendor
RPI1	BUZZER	BUZZ	LZL	Action	Ring the buzz with specific frequency and time		Time Alarms	
RPI2	Button	IsPushed	LZL	Report	Is the button pushed			
RPI1	LED	TurnOn	LZL	Action	Turn on the LED light		Lighting	
RPI1	LED	TurnOff	LZL	Action	Turn off the LED light		Lighting	
RPI2	Temp	ReadTemp	LZL	Report	Read the temperature.		Environment Monitor	

Figure Services: Example Services Tab

React App

File Edit View Window Help

Things Services Relationships **Recipes** Application

Logic
Services
Relationships
Literals

App name "Light and Buzzer App"

Recipe

if

then

RPI1LEDTurnOn
Input

RPI1LEDTurnOff
Input

RPI1BUZZERBUZZ

Input1 1000

Input2 10

Save

Open

Figure Recipes: Example Recipe Built. Note that the input field of services should be filled; I forgot to add when I screenshotted.

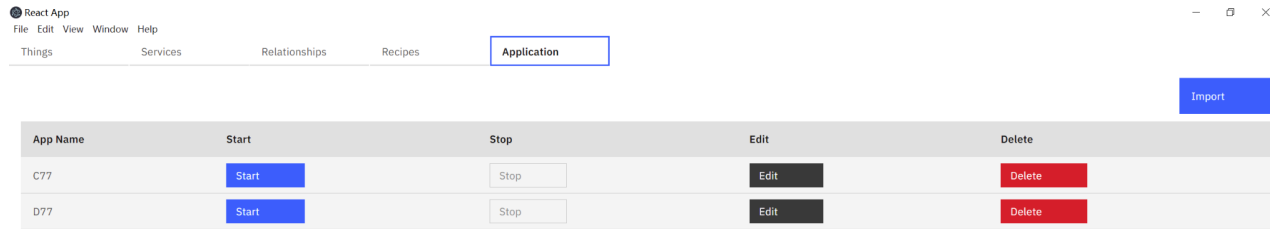


Figure Apps: Example App page, we can run, stop, edit, import, or delete an app in this tab.

3.1 Subjective Evaluation (0.5 page)

At the end of the current development phase, the Atlas IoT IDE project is able to correctly interpret IoT microservices masked by Atlas IoT DDL, “control”, “drive”, “support”, and “extend” IoT microservice relationship. In addition, the software is also capable of displaying currently available IoT Things, services, relationships, and applications within its GUI, allowing users to compose IoT applications graphically in the recipe tab, and saving or loading IoT applications. With a limited period of time to work on the project, we think the progress we made for this project is satisfying.

The main challenges for the project are adapting to new frameworks, handling asynchronous development, and resolving design misunderstandings with the architecture of the IoT development environment.

3.2 Future Work (0.5 pages)

- Support two more kinds of relationship

- Test the framework on more physical devices
- Support unbounded service in the IDE
- Extend the input type of services from integer to other more types
- Allow things to leave the smart space dynamically
- Allow trigger and time delay in application design (to realize functionality like turning on the alarm clock on 8'o clock or two hours later in a more easy way)
- Detect the IP environment and change it automatically instead of manually
- Generate executable files to run the app without the help of IDE

3.3 Distribution of Effort (who did what, names listed alphabetically)

Student Name	What was done?	% Effort	Comments
--------------	----------------	----------	----------

Li, Jiahao	Mainly developed Front End, Helped debug Back End	100%	
Li, Sijia	Mainly developed Front End, Helped debug Back End	100%	
Liu, Xuanning	Mainly developed Back End, Helped debug Front End	100%	
Zhu, Qiyue	Mainly developed Back End, Helped debug Front End	100%	