2022-07-02

Team Conductor
System Design Architecture

Table of Content

CRC Cards

Orchestra

Class Name: User	
Parent Class:	
Subclasses: Developer	
Responsibilities: ● Knows one's email and password (1) ● Knows one's connected devices (1) ● Has an identifier (userId) ● Adds/modifies one's connected devices (2) ● Removes one's connected devices (19) ● Installs bots from the marketplace (5) ● Imports/exports source code on the bot description (marketplace) (6, 10) ● Searches bots from the marketplace, using keyword or tags (9) ● Leaves ratings to the bot on the bot description (marketplace) (11) ● Writes comments to the bot on the bot description (marketplace) (11) ● Knows one's comments/replies on the bot description (marketplace) (12) ● Selects/deselects the devices with installed bots, given device's identifier (8) ● Removes the installed bots, given installed bot's identifier (16) ● Adjusts scheduling of installed bots (17) ● Generates events on the calendar (21) ● Modifies events from the calendar (21) ● Imports .ics files into calendar and manipulates events (22) ● Imports custom bots locally and have the web app display all locally created profiles/bots (20)	Collaborators: ● Device ● Bot (Installed Bot?) ● Comment ● Calendar (Event?)

Class Name: Developer	
Parent Class: User	
Subclasses:	
Responsibilities: ● Uploads self-made bots (executable file) on marketplace (4) ● Replies to comments on the bot description (marketplace) (12) ● Releases an updated version of uploaded bot (13)	Collaborators: ● Uploaded Bot ● Comment

Class Name: Device	
Parent Class:	
Subclasses:	
Responsibilities: ● Has an identifier (2) ● Knows its device name (2) ● Knows its device description (2) ● Knows its platform (2) ● Knows its connected bots (2, 8) ● Knows its device status (2, 18)	Collaborators: ● User ● Installed Bot

Class Name: Bot	
Parent Class:	
Subclasses: Uploaded Bot	
Responsibilities: ● Has an identifier ● Knows its bot name ● Knows its bot description ● Knows its version	Collaborators: ● User

Class Name: Uploaded Bot	
Parent Class: Bot	
Subclasses: Downloaded Bot	
Responsibilities: ● Has an identifier ● Knows its uploaded date (4, 10) ● Knows its updated date (10, 13) ● Knows its developer (uploader) (4, 10) ● Knows its source code link (6, 10) ● Knows number of downloads (4, 10) ● Knows its ratings and comments (4, 10) ● Knows its image-link (4, 10) ● Knows its tags (9, 10)	Collaborators: ● User (Developer?) ● Device ● Comment

Class Name: Installed Bot	
Parent Class:	
Subclasses: Uploaded Bot	
Responsibilities: ● Has an identifier ● Knows downloaded date (2, 5) ● Knows connected device (2, 16) ● Knows its version, given downloaded date (2) ● Knows its status results after execution (error, warning, or success) (15) ● Knows its connected events and jobs (21)	Collaborators: ● User ● Device ● Event

Class Name: Comment	
Parent Class:	
Subclasses:	
Responsibilities:	Collaborators:

● Has an identifier ● Knows its comment (11) ● Knows its writer, given userId (11) ● Knows its written date (11) ● Knows its parent-comment, so this comment can be regarded as a reply (12)	● User (developer) ● Uploaded Bot

Class Name: Calendar

Parent Class:

Subclasses: Event

Responsibilities:	Collaborators:
● Knows today's date (21) ● Knows all events (21)	● User

Class Name: Event

Parent Class: Calendar

Subclasses:

Responsibilities:	Collaborators:
● Has an identifier ● Knows its date/duration/time (21) ● Knows its event name (21) ● Knows its event detail (21) ● Knows its connected bot(s) (21) ● Knows its connected device(s) (21)	● User ● Bot (Installed Bot?) ● Device

Interface

DAO

Responsibilities:	Collaborators:
● Create a user, given unique email address and password (1) ● Get a user, given correct email address and password (1) ● Get all connected devices of a specific user (2, 17)	● User ● Bot (Uploaded Bot & Installed Bot?) ● Calendar (Event?)

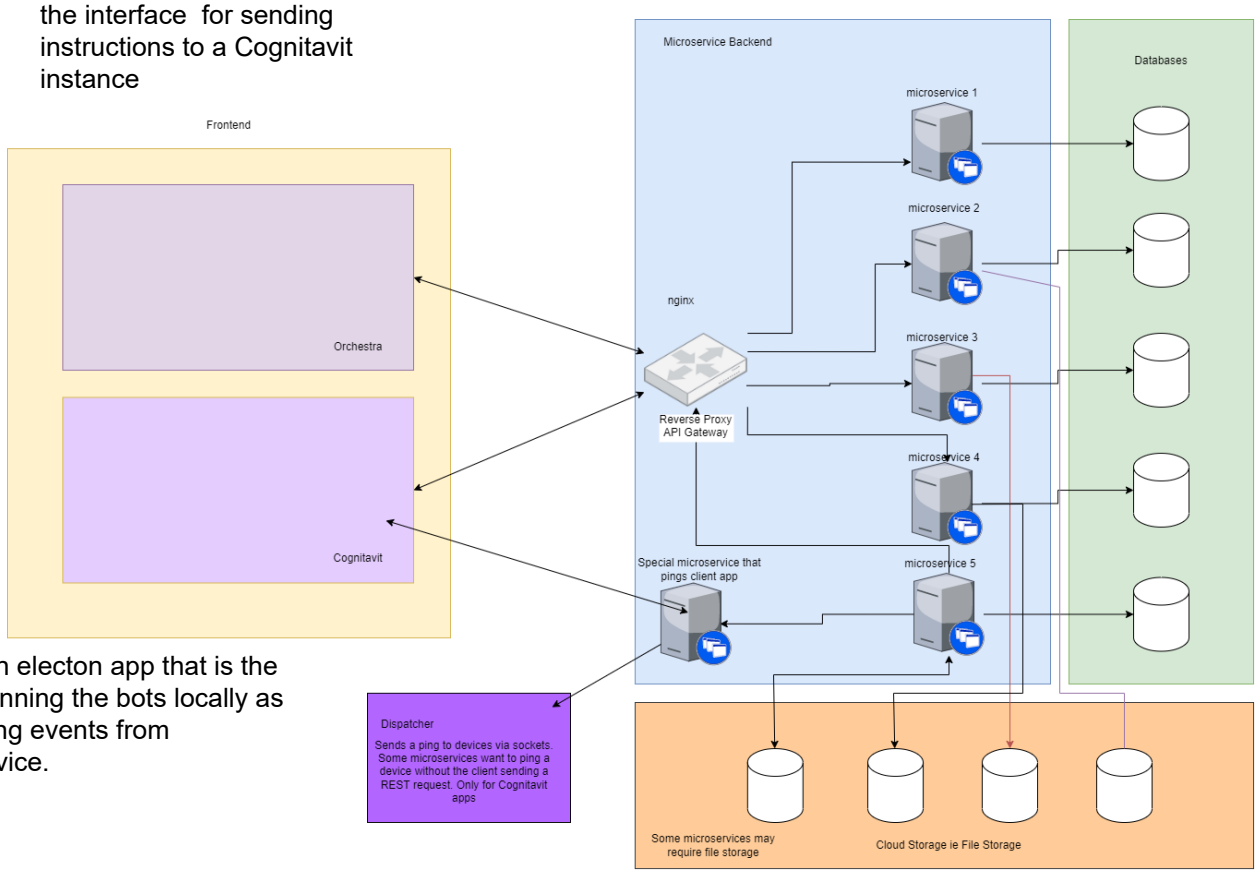
● Get details of connected device, given device identifier (2, 18) ● Modify details of connected device, given device identifier (2) ● Remove connected device, given device identifier (19) ● Upload self-made bots (executable file) on marketplace (4) ● Update uploaded bots from the marketplace, given userId and bot identifier (13) ● Install pre-made bots (executable file) from the marketplace (5) ● Get all uploaded bots, given search keyword or tag identifiers (4, 9) ● Get details of uploaded bot, given its identifier (6, 10) ● Get comments of uploaded bot, given bot's identifier (11) ● Colorize developer's comments of uploaded bot, given userId (12) ● Post comments to the uploaded bot, given userId (11) ● Get all installed bots, given userId (16, 17) ● Get details of installed bot, given its identifier (15, 17) ● Connect/disconnect installed bot and device, given bot identifier and device identifier (16) ● Delete installed bots, given bot identifier (16) ● Get a summary of all connected devices and installed bots (17) ● Import custom bots, given custom bots (22) ● Get all events of selected month, given month (21) ● Creates a new event (21) ● Modifies an existing event, given event identifier (22) ● Import external calendar, given .ics files (22)	

Software Architecture Diagram

Orchestra is a React.js app that runs in the browser and is the interface for sending instructions to a Cognitavit instance

the nginx acts as an endpoint for communicating between microservices

the microservices provide the various features of Orchestra and each of them are miniature for full-stack application



Cognitavit is an electon app that is the interface for running the bots locally as well as receiving events from dispatcher service.

binaries are stored on an FTP server. they are fetched by cognitavit and installed on the local machine

System Architecture

The Software Architecture for our project is a combination between microservice, and three tiered architecture. We are using nginx web server which is acting as a reverse proxy to microservices. This allows us a single endpoint for a client to connect to . In the backend each microservice communicates to each other via REST API. In terms of design and implementation, we will try to avoid communication between microservices, as it reduces our ability to be resilient to network failures and makes our scalability more difficult. Our microservices are written in express.js. Some microservices may require

https://microservices.io/

https://www.ibm.com/cloud/learn/microservices

https://www.ibm.com/cloud/learn/three-tier-architecture

System Decomposition

The users have 2 ways to use our application, either via the Web Application, or the Client side Application. Both will communicate to the backend via RESTful api. Failure in the backend means that it's within a specific microservice, this means that the entire application will not crash and fail, except the specific feature the microservice that stopped working provides for. Database errors are also specific to a microservice, since each microservice does not share a single database. User input is checked via the client side before being sent to the server side. If needed our backend will have a series of middlewares that perform checks on the input before passing it to the backend code. This allows us to catch errors and reject the input with an error 400 code.