CP490: Welcome to K Division

March 29, 2021

Blackboard Collaborate 0930h

Mr. Klein // dklein@confederationc.on.ca

Contents

Overview

Congratulations on joining K Division! We're a brand new startup producing interactive, yet oddly anachronistic user experiences in the cloud. In your new role at the company, you'll be developing a whole new platform for experiencing late 70s and early 80s interactive fiction.

You will be split into at least three (3) teams. Each team will own a specific part of the application and you will be responsible for designing, programming, and deploying it.

You will need to interact with the other teams in the class to ensure that all parts of the application work together. You will also need to interact with the product owner (me) to ensure that your deliverables meet requirements. Finally, a fictional operations team will also stand up your application, so your team will need to support them by producing documentation and answering questions.

Course Format

The class will meet on Blackboard Collaborate, as a whole, 3 times per week. Meetings will be held Monday, Wednesday, and Friday (except for Good Friday). These hour long meetings will start at 9:30am EDT.

Class meeting time will be spent on:

- **Team status reports (5 minutes per team, ∼15 minutes)**: Each team will share a summary of their activities – what the team worked on, what's up next, and challenges.
- **Architecture / Design / Document Reviews (30 minutes)**: Teams will present their solutions to the group for feedback and review. While not formal presentations, teams should be prepared to discuss the technical details of the problems they are solving. Teams should be prepared with any diagrams and supporting reference material so that the class is able to understand and contribute feedback. Discussion is to be focussed on gaps and potential trouble areas.
- **Problem solving or bootstrapping (15 minutes)**: This time is allocated to help teams solve problems, whether of a technical nature or in coordinating deliverables between teams. This project is likely to use technology unfamiliar to the class. I will lead sessions to cover this material.

Lastly, I will meet with individual teams on an as–needed basis. I'll schedule those meetings with the team, at our mutual convenience, during "business hours" (10am — 4pm).

Onboarding

This document is available on Blackboard. Outside of that, course content is hosted on Github and Slack.

We will be using Slack to communicate. Use this invite link to join:

`https://tinyurl.com/4a469zjr`

Github is where teams will host their code and manage project tasks. Please email me your Github account name *as soon as possible* so that you can access the project boards and repositories:

`https://github.com/cp490-kdivision`

We will review in class how these tools will work for us.

Evaluation Criteria

This class is PASS / FAIL, so no graded marks are given. A PASS in this class is earned by demonstrating that you provided value to your team and helped achieve its goals, while meeting the *General Expectations* outlined below. Individuals who do not contribute technically, are unable to work within agreed-upon processes, and cannot coordinate with other teams will not achieve a PASS.

To aid in my evaluation, you will write a summary of your individual contributions to the project.

General Expectations

I will hold you to the same standards that I have for junior development staff. That is, I know you're capable of doing the work, and you will need help learning the tools and processes.

While I hope that the class as a whole produces a working application in which everyone can take pride in, it's entirely possible that it won't come together as expected. That's ok – my goal is that you work within software processes, talk with other teams, debate your ideas, and learn new tools.

To that end, we will operate according to these values:

- **Respect always**: Assume that people are acting in good faith. Earn the trust of your teammates and give it, too.
- **Fundamentals first**: Value the basics of testing, designing, and communicating. Creating complex solutions is possible through the diligent, regular application of focussed effort. Know your tools and practice your skills.

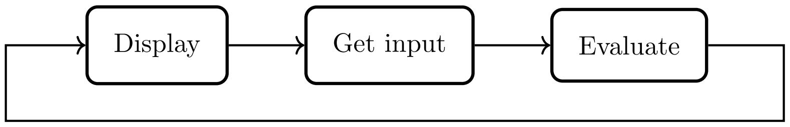

Figure 1: Core game loop

- **Start simple and iterate**: Value taking action above waiting for answers. Reach out instead of disengaging. Sharing a partial solution today is far more valuable than trying to have everything working next week.

Project

K Division creates software to create and play multiplayer online text-based role-playing games using their proprietary, leading–edge COAL (**C**oncurrent **O**nline **A**dventure **L**and) Engine. Another name for these kinds of games is *Multi-User Dungeon*, or MUD.

Since K-Division is a startup, the game engine isn't written yet, so some of you are going to need to do that. They don't have a front-end yet, either. Or a database backend. It's not much of a company, really, but the investors are thrilled!

Your job here is to turn this vision into reality.

Background

The MUD, and its single–player cousin, the Text Adventure, were precursors to Massively Multiplayer Online Roleplaying Games (MMORPGs). In a MUD, players connect to a server using a command line terminal, read descriptions of the environment their characters are in, and type commands. Those commands allow the player to travel, engage in combat, manage inventory, craft, and interact with other players.

If you haven't played one of these, then you might want to try Achaea, which is a rather more sophisticated version of what we're trying to create.

Playing a MUD

In a MUD, a user takes on the role of a character that wanders around a virtual world, solving puzzles, battling monsters and other characters, amassing various items, and conversing with other users.

The user is presented with a text description of their character's current location and a command line prompt. For example[1],

```
West of House
You are standing in an open field west
of a white house, with a boarded front
door.
There is a small mailbox here.

>
```

You can see the various parts of text:

- **title**: West of House
- **description**: You are standing in . . .
- **item list**: A small mailbox
- **prompt**: The '>' character

Descriptions may also include lists of:

- valid exits, insofar as the player character can see
- other player characters in the same place

The user then types in some text. For example, `GO EAST` or `OPEN MAILBOX`.

The game processes the input and replies with a new description, and another prompt. That is the core interface and game play; see Figure 1:

- **Display**: The game presents a block of text
- **Get input**: User provides input, also text
- **Evaluate**: Game engine processes the incoming text, updating the game state

Game Objects

There are five types of objects in the COAL engine:

- **Rooms**: The locations inhabited by the player characters
- **Items**: Things that player characters interact with such as mailboxes
- **Players**: How individual users are represented in the application
- **Characters**: The internal representation of a player character
- **Events**: How the game engine responds to user commands

Figure 2 shows how these objects relate to one another. Games consist of characters, items, and rooms. Players command characters in a game. Both players and rooms can possess items. Finally, characters move from room to room.

[1]This is the opening of *Zork I: The Great Underground Empire* (Infocom, 1980)

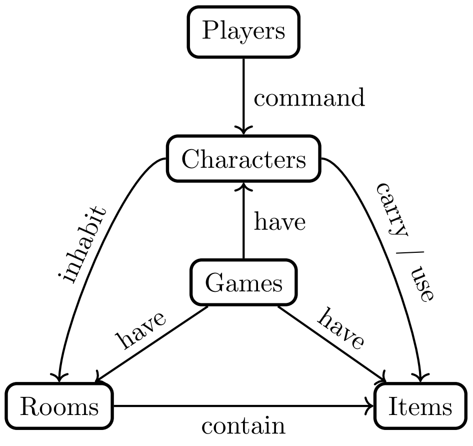

Figure 2: COAL game objects

Property	Description
Title	Short name for a room, e.g. `Forest`
Description	Longer text with details of what the room looks like
Exits	A list of visible directions mapped to destinations, e.g. `East` → `House`
Attributes	A list of key–value pairs for state information, e.g. `DARK` → `True`

Table 1: Room properties

Each object type has a number of *properties*. Those properties are either *mutable*, meaning that they can be changed as part of playing the game, or *immutable*. Properties that are immutable can only be edited by game authors outside of the game and cannot be changed through player interactions directly.

Rooms

The locations in the game are collectively referred to as rooms, even though they could be describing outdoor locations, caves, spaceships, or more abstract settings[2]. Properties of rooms are given in Table 1. When a user enters a room, the full description is shown. A list of directions, like `NORTH` or `UP` are mapped against other rooms, connecting them. This list of exits is usable by, and shown to, the player. The title, description, and exits are all immutable properties. The attributes property is mutable, and can be used to track game state that

[2]For example, A Mind Forever Voyaging (Infocom, 1985)

Property	Description
Name	Short name for the item, e.g. `Sting`
Aliases	Other words that can be used for the item, e.g. `sword`
Description	Additional details, e.g. `An Elvish short-sword that is somewhat fond of spiders`
Attributes	A dictionary of key–value pairs for metadata (for example, to track the number of times a potion can be used, and item might have: `doses` $\rightarrow$ `5`)
Location	Current room the object is in (or none, if it's inside something else or in a player's inventory)

Table 2: Item properties

Property	Description
Title	Name the user has chosen for an account in the COAL application
Attributes	A dictionary of key–value pairs for storing information about players

Table 3: Player properties

pertains to the room.

Items

Items are what users manipulate in the game: weapons, armour, keys, treasure, and so on. The properties of items are given in Table 2. An item has a name, but also possibly aliases. For example, there may be a book with the name *Hitchhiker's Guide* that also has an alias of *guide*, so the user can type: `GET GUIDE` or `GET BOOK` interchangeably. Items may have other attributes, too. For example, if the game has combat elements, then items could have attack or defense values.

Players

A player object represents a user in the COAL application. Table 3 shows the properties of the player object.

Property	Description
Title	In–game name of the player character
Player	User account the character belongs to
Inventory	Items acquired by the player character
Attributes	A dictionary of key–value pairs for storing game state, e.g. `HAS-SEEN-WELCOME-TEXT` → `TRUE`

Table 4: Character properties

Property	Description
Command	Static text string describing the syntax of the action. See the *Input Parsing* section for examples.
Conditions	A list of assertions that govern whether the *True items* or *False items* should be run in response to the command.
True items	A list of game engine functions that manipulate the game state if all of the event conditions are all met, or if there are no event conditions.
False items	A list of game engine functions that manipulate the game state if any of the event conditions are not met.

Table 5: Event properties

Characters

Table 4 lists the properties of player characters. These are the avatars for the user – the character they play in the the game, and so all the state in this object is mutable. Attributes allow the game engine to track game state. For example, the player character may be poisoned, or under the influence of a potion that lasts for a given number of moves. This key–value mechanism provides an open–ended way to create and manage character data. Each character is associated with one and only one player account, but a player account may have multiple player characters.

Events

The properties of events are given in Table 5. The game logic is described, by game authors, using events. All the properties are immutable. The game engine considers each event command string as a pattern; how this works is discussed in the *Input Parsing* section. Once the engine has has determined which event to execute, all of the conditions for that action are evaluated. If they are all met

Property	Description
Primitive	A game engine function. For example, to display a message to the user, the game engine defines a function called `message`.
Arguments	Arguments to the game engine function. For example, which message to display.

Table 6: Event item properties

(or if there are no conditions), then the true actions are evaluated. Otherwise, the false actions are evaluated. The true or false actions dictate actual changes to the state of the game – moving characters from room to room, manipulating items, and any other game mechanics. The conditions and actions have the same format, given in Table 6. A *primitive* is an actual function in the game engine. Content creators will need to have a list of these primitives in order to author the game. The arguments given to the primitives are strings, but can be constants or variable names that are part of the command definition.

Creating Game Content

Game play requires game content. In many MUDs, once you attained a certain rank within the game[3], you were allowed to author game content. In other early or small MUDs, updating game content meant recompiling the source code. The prevailing idea, however, was that you could add new content as meta–commands in the game.

For this project, game content creation is separate from game play, but an integral part of the COAL application. Users should be able to create games, rooms, items, and events. The API is designed with the idea that all game content can be authored in the system. Further, the system exposes a list of functions corresponding to the *primitives* discussed in the previous section. Users designing a MUD should be able to take the list of primitives and author a complete game.

Design and Architecture Principles

Cloud computing is a popular way of deploying large-scale enterprise business applications. Companies large and small outsource their infrastructure and host their applications with cloud computing companies – be it Amazon, Microsoft, Google, or others. One reason for Amazon's success as a cloud provider is their emphasis on creating *platforms*. The same website that let them sell books

[3]sometimes called wizard status

became the same one that lets everyone else sell, well, everything. It's a *platform* for online sales. Further, Amazon Web Services (AWS) is consumed by Amazon internally, meaning that the services Amazon sells are the same ones it uses to run their business.

Similarly, the K Division COAL application is a *platform* for MUDs. The game that we should be able to play at the end of this is a *demonstration of the platform* more than a shrink–wrap product[4].

The way we build platforms is through *Application Programming Interfaces* (APIs). APIs are everywhere. Google has all kinds. Zoom? They have an API. Do you use Slack? They have an API to write bots. So does Facebook. Maybe you play games and use Discord. Here's their API.

An *API* is a kind of programming contract. If I provide your service with input that conforms to your specification, then your service agrees to return a result in a given format. For example, an API can be written to accept and return JavaScript Object Notation (JSON) objects. To describe APIs, we are going to use OpenAPI documents. OpenAPI is a way of writing descriptions of API contracts in a way that lets us use other code generation software tools to create clients and servers for us.

You will be given an OpenAPI document that specifies the public–facing API of the COAL Engine. You will negotiate internal APIs with your peer teams. I expect to be able to run a sample game client against this public API and use it to explore your work.

You may want to negotiate changes to the public API, if it doesn't let you fulfill your requirements. Any changes will be discussed in the full class meetings.

Application Architecture

The overall application architecture is given in Figure 3. The class is responsible for implementing all of the services shown in the unshaded boxes. The *Command Line Interface (CLI) Client* is provided, and the third–party *Slack* integration is an example here; see the list of *Secondary Objectives*.

The application consists of the following layers:

- **Presentation Layer**: This layer, consisting of all of the Services and clients above the Proxy service, implement the functionality that lets users play and create content for COAL Engine games. The Web Service makes public API calls into the Proxy Service and presents the results to the user.
- **Application Logic**: The Proxy service brokers incoming client requests, translating them into calls for other services. It exposes the public API, as given in the OpenAPI specification. The Player service provides user

[4]read more about this here: `https://gist.github.com/chitchcock/1281611`

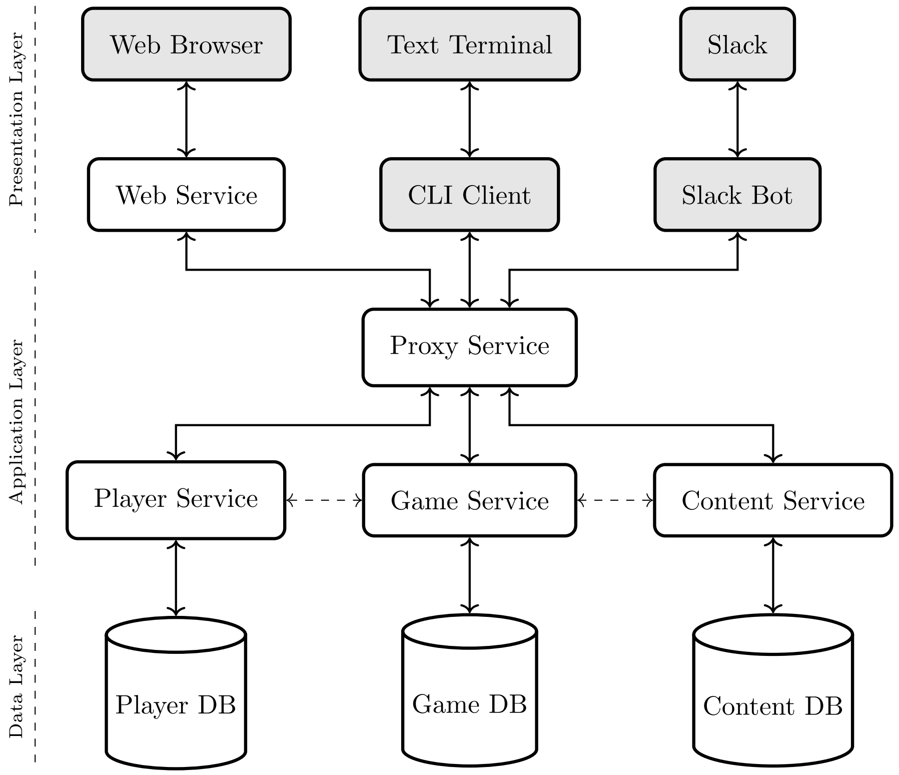

Figure 3: Overall Application Architecture

and player character support functions. The Game service implements the core game loop, evaluating user input and calling into the other services to update game state. The Content service manages game assets such as room and item descriptions, and provides it to the Game service.
- **Data Layer**: The player, game, and content services manage their respective portions of the COAL engine. Since each of these services have different responsibilities, they each have their own databases and associated schemas.

The components of the system are described below, beginning with the REST API itself.

Public REST API

A REST API is a way of describing objects and operations against them. A reasonable set of design principles for REST is given here. In brief, the API has as set of *resources*, or objects in the system, and *methods* used to manipulate them.

The COAL public API is given in the API Specification PDF. Each URL endpoint is given, along with the expected input parameters, HTTP verb (GET, POST, PUT, or DELETE), response code (e.g. 200, 201, 404) and output format. All data is transferred via JSON documents.

For example, a *game* is a resource. Each instance of a *game* resource can be completely different – different game logic, different rooms and items. Having different games is how COAL is a platform instead of just a game engine.

To list all of the available games in the COAL application, the `/game` endpoint is used with the HTTP `GET` verb. For example, consider a Proxy service that is listening to port 8100 on the local machine. Using the cURL tool, the list of games can be retrieved like so:

```
; curl -X GET http://localhost:8100/v1/game
[
  {
    "id": "7c6b1fc3-0919-495c-bc80-dc1a8737a745",
    "title": "undermud"
  }
]
```

A JSON document is returned by the Proxy service. That document is a list of dictionaries. In the one dictionary that was returned, the short title of the game, `undermud`, was given, along with an internal unique identifier.

To retrieve the full details for a resource, the identifier for the resource is appended to the URL. For example, to retrieve the full game details, the `id` above is appended to the URL:

```
; curl http://localhost:8100/v1/game/7c6b1fc3-0919-495c-bc80-dc1a8737a745
{
  "description": "MUD for all",
  "id": "7c6b1fc3-0919-495c-bc80-dc1a8737a745",
  "attributes": {
    "starting-message": "Welcome to UnderMUD!",
    "starting-room": "56065d49-0499-43ec-a0aa-b9e74f91a198"
  },
  "title": "undermud"
}
```

The `id` and `title` are returned, along with the long form description and attributes.

Creating a game uses a different HTTP verb: `POST`. A JSON document needs to be provided that specifies the title and description for the game:

```
; curl -X POST \
       -H 'Content-type: application/json' \
       http://localhost:8100/v1/game \
       -d'{"title": "new", "description": "New MUD!"}'
{
  "description": "New MUD!",
  "id": "dd290617-a480-4285-a857-e3312a3be831",
  "properties": {},
  "title": "new"
}
```

In the cURL call, the HTTP verb (`POST`) is specified, that the data is a JSON document (`-H 'Content-type...'`), and the data itself. The Proxy service sends the request to the Game service, which creates the game and sends the details back. The `id` is generated by the Game service.

Other game objects correspond to sub–resources of the `/game` endpoint:

- `/game/{game_id}/room`: Manipulate rooms; see Table 1
- `/game/{game_id}/item`: Manipulate items; see Table 2
- `/game/{game_id}/event`: Manipulate events; see Table 5 and Table 6

Users have accounts with the COAL application in the form of player objects. Signing up with the application is a call against the `/player` endpoint. While characters can be retrieved as a sub–resource: `/player/{player_id}/character`, creating a character is a function of the game service.

That is, creating a character is a POST to:

`/game/{game_id}/player/{player_id}/character`

The game service is responsible for creating the character and establishing any initial attributes.

The API Specification PDF describes all of the endpoints that the Proxy service must implement. The specification details all of the JSON documents that are sent and received, including fields names and structures. The specification is also what game clients (for example, the Web service) must implement in order to properly communicate with the Proxy service.

OpenAPI Specification

The API Specification PDF that you are given describes in detail the COAL REST API we began to explore in the previous section. The text file used to create the PDF is also given to you, and is called `openapi.yaml`. This file is the OpenAPI document mentioned in the *Design and Architecture Principles* section. The document is a text file in YAML format; see this page. Describing APIs in this way lets us use tools to generate code for servers and clients in a large number of languages. These YAML documents are also used in testing services and clients.

Using the same example as above, retrieving the details of a particular game from the COAL application is described by the following fragment:

```
/game/{game_id}:
  get:
    summary: Retrieve game details
    parameters:
      - $ref: '#/components/parameters/gameId'
    responses:
      '200':
        $ref: '#/components/responses/GameDetailResponse'
```

The path (`/game`) is first, followed by the HTTP verb: `get`. The URL parameters are listed after a human–readable description. The `$ref` is a *path* to another part of the document that describes the `game_id` identifier.

Expected responses are described starting with their HTTP status code. Since responses can be the same for different operations, describing the response only once avoids errors. The `$ref` allows sections of the document to be re–used. The `GameDetailResponse` contains these details:

```
responses:
  GameDetailResponse:
  description: Game detail
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/GameDetail'
```

The `content` section shows that the response should be in JSON format, and that it should contain a `GameDetail` object, shown here:

```
GameDetail:
  type: object
  properties:
    id:
      type: string
    title:
      type: string
    description:
      type: string
    attributes:
      type: object
      additionalProperties:
        type: string
```

Finally! The `GameDetail` object describes the properties that will be returned in the JSON document. Each property (`id`, `title`, ...) has a data type, at a minimum, but more sophisticated definitions are possible (see: the OpenAPI specification). Each of these properties lines up with part of the JSON from the example in the previous section.

The OpenAPI YAML file is quite long. The API Specification PDF is meant to help you interpret the specification in terms of the JSON that is sent and received by the services. Additionally, you can use the tools listed at the end of this document to explore the YAML further.

If a team wants to propose changes to the public COAL API, then I will work with the team to present the modifications necessary to the OpenAPI document.

Services

The COAL Application consists of a number of *services*. A service is a program can be deployed (installed and configured) on a computer (virtual or physical). The service can be started, stopped, and tested independently of the other services in the application. Each service has an API that is used to communicate with it.

The following sections describe the purpose and scope of each of the services.

Some of the descriptions describe a data model that the service will need. The `title` data model field is a unique, human–readable, one word **key**. Think of a `title` like a hashtag. In contrast, `description` fields are potentially multi–line text fields with as much text as needed. Properties tables are dictionaries or maps of keys (`titles`) and values; they're an open–ended mechanism for storing data. Tables have a unique primary key generated by the database – the `id` field. This is called a surrogate key, and has significant practical advantages over having the `title` field be the primary key.

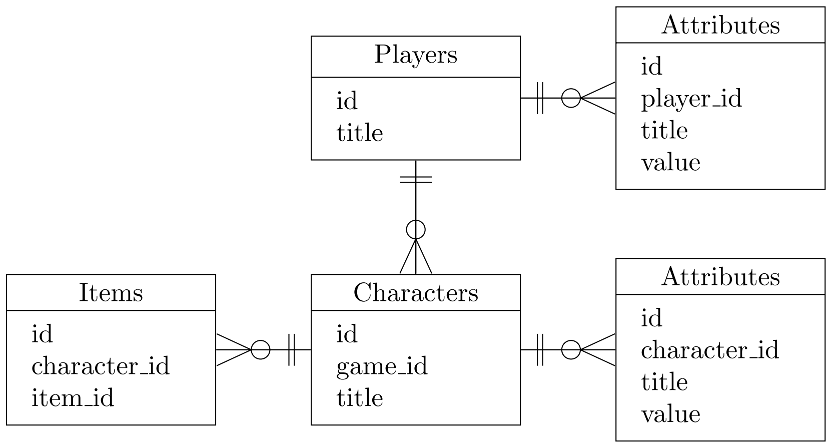

Figure 4: Player service data model

Web service

The web service provides the front–end for the game. Users must be able to manage their account and game characters. They must also be able to play the game itself using an interactive console. While the Web service renders the game output, it uses the public API to communicate with the Proxy service.

Proxy service

The Proxy service is a bridge between the public API and all of the internal components of the application. It has no direct access to the databases, and does not know how the data is stored. Instead, it accepts incoming requests and translates them into messages for the internal services and then sends back properly formatted responses.

This design is a pattern for providing *horizontal scaling.* Multiple Proxy services can be launched to accommodate as many clients as necessary, independent of the other services, since those services may themselves have specific ways to manage their resources.

The Proxy service must send and receive JSON content according to the COAL public OpenAPI specification.

Player service

The Player Service is responsible for user accounts and managing the player characters the user creates. Figure 4, shows a basic data model. Players may have multiple characters. Each character has a set of properties and a set of

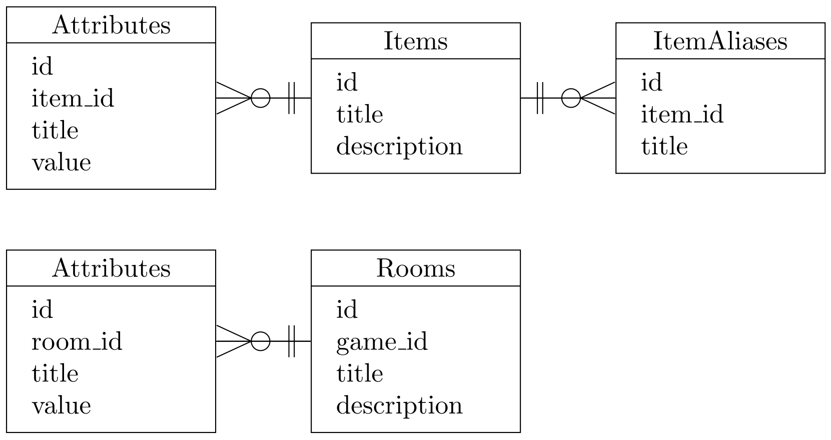

Figure 5: Content service data model

items. The `item_id` and `game_id` foreign keys belong to the content and game services respectively.

Content Service

The Content service contains all of the game assets – text, descriptions, and properties for rooms and items. Figure 5 is a basic data model. Items can have multiple properties and aliases. An important attribute for rooms is their exits. The direction (for example, `NORTH`) would be the attribute `title` and the title of the destination room would be the `value`.

Game Service

The Game Service is responsible for managing game play. The service reaches out to the content service for room and item descriptions, and to the player service to fetch and update character data. Figure 6 has a basic data model. The game engine parses user input (described in the next section), evaluates events, and returns results back to the user.

Locations of characters and items are tracked by the Game Service. Consolidating this game state information makes transaction logic easier to manage and ensures that item content remains immutable. In this model, either the `room_id` or the `character_id` is used in a table row, but not both.

The game engine logic is captured in the `Events` table and its children. An event has three parts – conditions to evaluate, steps to follow if those conditions are met, and steps to follow if they do not. All three of these parts consist of ordered lists, and so have a `position` field. Also, all three of these parts consist

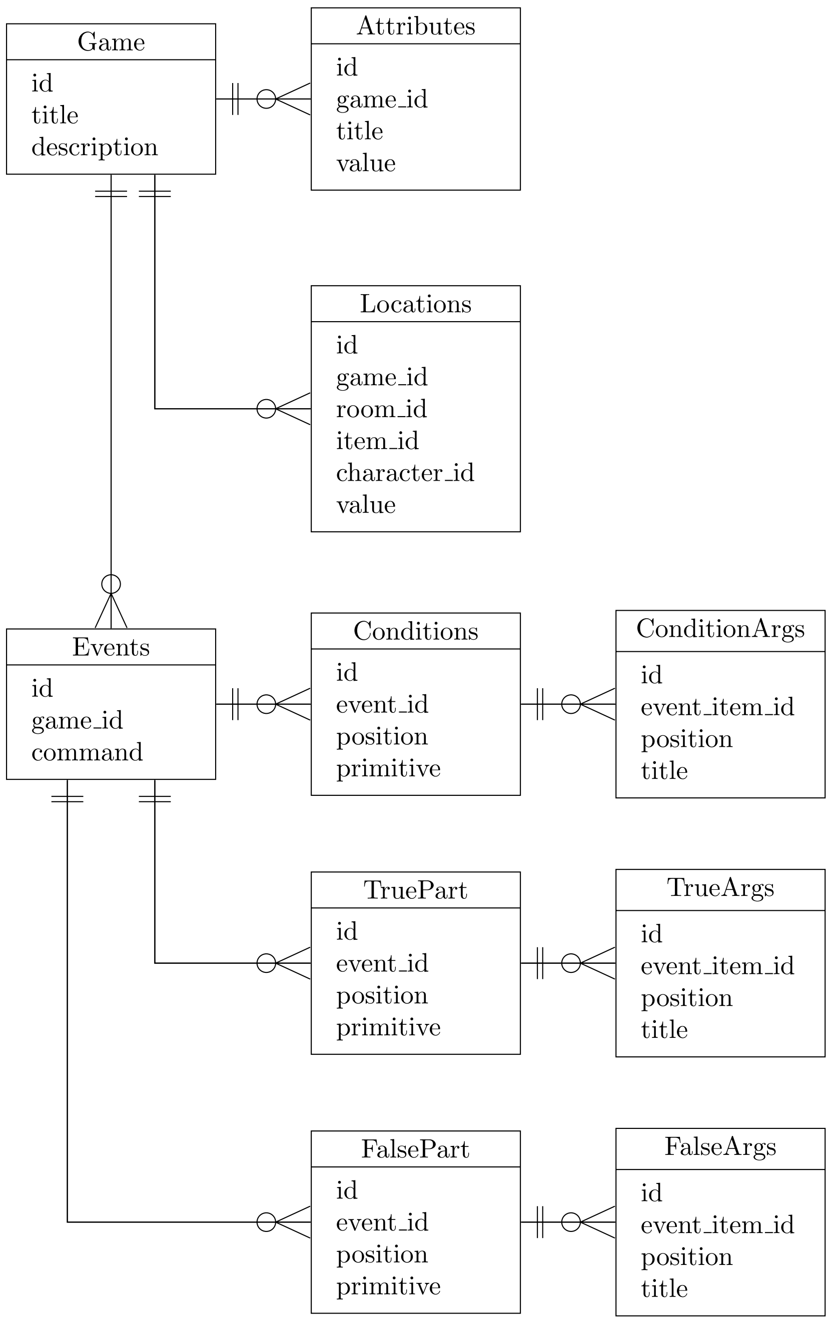

Figure 6: Game service data model

of a game engine function, called a `primitive`, and arguments to that function captured in the `ConditionArgs`, `TrueArgs`, and `FalseArgs` tables.

Input Parsing

Text adventure games and MUDs have all kinds of ways to understand what the user is trying to do with their characters. Turning user input into actions for the game engine is called *parsing* the input.

The COAL engine uses a specific parsing algorithm that's not as sophisticated as some[5], but also not as simple as the ones used in the first text adventure games[6].

Consider that each `Event`, as above, has a `command`. That `command` is a pattern consisting of words and variables. For example, all of these could be commands in the same game:

- `GIVE !ITEM TO !CHARACTER`
- `GIVE !ITEM TO BEAR`
- `GIVE HONEY TO BEAR`

In the first command, there are two variables: `!ITEM` and `!CHARACTER`. All variables are prefixed with a (`!`). If the user typed: `GIVE LANTERN TO WIZARD`, then that would be a valid match for that command. However, if the user typed: `GIVE FOOD FOR BEAR`, then that wouldn't be understood, because `FOR` isn't part of any of the command patterns.

Algorithm 1 COAL Parser, Part 1

```
procedure FindEvents($input$)
    $w \leftarrow$ number of words in input
    $c \leftarrow$ commands of length $w$
    for $i \leftarrow 1, w$ do                                    ▷ Look at each word
        $keepers \leftarrow$ empty list          ▷ Store events that match the input word
        for $r \leftarrow 1, c$ do                                ▷ Check each event
            if $w[i] = r[i]$ or $r[i]$ is a variable then
                $keepers.append(r)$
            end if
        end for
        $c \leftarrow$ keepers                         ▷ discard events that do not match
    end for
    return $c$                               ▷ return all events that match the input
end procedure
```

[5]For example, the parser used by the Text Adventure Development System (TADS)

[6]Specifically, the first commercially successful ones published by Scott Adams in 1978, which understood only two words, and only the first three letters of each word at that.

The first part of the algorithm is given in Algorithm 1. This part creates a list of matching events. First, the database is queried for all events that have commands with the same number of words as the input. Next, each word in the input is evaluated. The input word is compared to the command words, in the same position, and events that match are kept. If the command word is a variable, then it's an automatic match – any input value is accepted. Otherwise, the input and command words must be the same for a match. Last, only events with matches are kept when looking at subsequent words. Once all of the input words are evaluated, the algorithm returns a list of events that match.

Using the algorithm, and the commands above, if the user typed: `GIVE LANTERN TO BEAR`, then the first two events would match. The first matches because both variables accept the input, and the second matches because of the `BEAR` part. The last rule does not match, because the user typed `LANTERN` and not `HONEY`.

If there are no matches on the events, an error is returned. For example, `I DO NOT UNDERSTAND` could be return as a message to the user.

If only one event is left, then that event can be evaluated (see the next section on how events are evaluated).

If multiple rules match, then the most specific one is selected. That is, an algorithm should find the event that has the fewest variables. In the example, that is this rule: `GIVE !ITEM TO BEAR`, since this has one variable, and the other rule has two (`!ITEM` and `!CHARACTER`).

Algorithm 2 filters the events from Part 1. Again, this algorithm searches word by word through the user input. All the rules will match – but if there's a word that matches *exactly* then that event is kept. At the end of the inner loop through all of the events, one of two things is true: (a) there is a set of events with exact matches, or (b) that particular word is a variable in all events. If (a) is true, then the set of events to process *becomes* the set of matched events. Otherwise, the algorithm continues onto the next input word.

At any time, if only one event remains, then the algorithm stops and returns that event. If, at the end of the algorithm, there are still multiple matches, then this represents a problem with the rules themselves – an ambiguity has been created. The program shouldn't crash at this point, though, but rather return an error.

Once an event has been found, it is evaluated. This evaluation can change game state and produces an output message for the user. The next section details this process.

Game Engine Primitives

Once user input is parsed and an event identified, it is evaluated. Game authors need to have the ability to make the game respond to commands, and the facility

Algorithm 2 COAL Parser, Part 2

```
procedure ReduceEvents(input, events)
    $w \leftarrow$ number of words in input
    for $i \leftarrow 1, w$ do
        if length($events$) $= 1$ then                    ▷ Is the algorithm done?
            return $events[0]$
        end if
        $keepers \leftarrow$ empty list
        for $r \leftarrow 1, events$ do
            if $w[i] = r[i]$ then                         ▷ Is the input word an exact match?
                $keepers.append(r)$
            end if
        end for
        if length($keepers$) $> 0$ then                   ▷ Keep only exact matches, if possible
            $events \leftarrow keepers$
        end if
    end for
    if length($events$) $> 1$ then
        return "Error"                                    ▷ Input is still ambiguous
    end if
    return $events[0]$
end procedure
```

Primitive	Arguments	Description
is_property_eq	key, value	Is the character attribute *key* equal to *value*
is_property_gt	key, value	Is the character attribute *key* greater than *value*
is_property_lt	key, value	Is the character attribute *key* less than *value*

Table 7: Event condition primitives

for this is another API. This API manipulates game state, but not content. That is, the API can set properties, but not change room descriptions.

The API consists of functions and arguments to those functions in the game engine. Arguments can be constant strings or command variables.

There are two classes of functions – conditions, and actions. Conditions are used to evaluate the game state. For example, `is_property_eq(key, value)` asks whether the value of character property `key` matches an argument. Each condition has a boolean result. A list of condition primitives is given in Table 7.

The first part of evaluating an event is to evaluate all of its conditions. If all of its conditions are true, then the list of true actions are run (see Table 6).

Primitive	Arguments	Description
message	key	Send the value of global attribute *key* to the user
set_key	key, value	Set character attribute *key* to *value*
look	none	Send a message with the the title, description, list of exits, and items of the character's current room

Table 8: Event action primitives

An event may also have *no* conditions, in which case the true actions are run automatically. Otherwise, if any condition is false, then the false actions are run.

A user can issue a command, have it recognized by the engine, but still result in some error message because of how the event conditions were evaluated. This is perfectly reasonable – `GIVE HONEY TO BEAR` shouldn't result in a successful result if, for example, the `BEAR` is not present or the character is not carrying `HONEY`.

An action affects the game state. For example, `message(key)` sends the value of the game property `key` back to the user. Some actions are given in Table 8.

Lastly, there is one set of events that are always evaluated. Events may have a command string that is empty. These are global events that run for every character on every input. An example of this is displaying a welcome message to new characters. A global event is defined that checks to see if the `first-time` key is set for a character. If it is, then nothing happens (the `true_part` is empty). If it isn't set, then: (a) the welcome message is sent, and (b) the `first-time` key is set.

Secondary Objectives

There are things missing from this specification. If we get to them, then these would be interesting:

- **Engine Primitives**: The lists of actions and conditions is most likely entirely incomplete. I haven't thought enough about authoring a game to work through what a *minimal* set looks like, so I've taken some guesses. If we get to the point of actually getting game content together then it'll make for some good discussion.
- **Access Control**: Content creation is normally reserved for the people running the game and their designates. The services should be extended to recognize different user access levels – player and admin, and restrict all of the content creation endpoints to be admin-only.
- **Non–Player Characters (NPCs)**: A feature of some MUDs are monsters and combat mechanics. Sometimes, there are shop and equipment

buying mechanics, too. While this might be possible using item objects, having NPCs as their own content type might be easier.

- **Character history**: Keeping track of everything a character has done – inputs, and the responses given by the application, is useful for debugging and user support. This feature would need updates to the data model, game engine, and some way to query the information. This feature may be useful to expose in the public API, too.
- **Third Party Integration**: The power of having a proxy service is in being able to integrate with other text platforms. For example, a bot could be written that facilitates playing COAL Engine games in Slack or Discord.

Tools and Technology

Implementation Language and Environment

What language the services are written in and the tools you use to do that are up to you and your team. Different teams may use different languages.

For what it's worth, however, there is a proof of concept implementation written for this project and it:

- was written entirely in Python,
- used SQLite as a database backend,
- lived on a Microsoft Surface Pro 6 running Windows 10,
- ran on Windows Subsystem for Linux v2 (Ubuntu),
- was debugged in Visual Studio Code.

Hosting

Standing up the application and seeing it work over the Internet makes all of this seem more real. Depending on the technology stack your team chooses, there are different places where you can host it for no cost. A caveat: while the services that you'd want to use are *free*, the company may still want a credit card to be able to sign–up.

For example, if you want a plain Linux server, then a cloud–based virtual machine (compute instance) would work:

- Amazon Web Services (`https://aws.amazon.com/free`) has a 12 Month Free EC2 compute instance option with 750 hours (roughly a month)
- Oracle Cloud (`https://www.oracle.com/ca-en/cloud/free`) has an always–free tier with 2 small Linux instances

You could use an app hosting service, like Heroku (`https://www.heroku.com/pricing`); the free option would also be fine for this course.

OpenAPI

To edit or visualize the COAL specification YAML file, use this site: `https://editor.swagger.io/`

The full specification for what OpenAPI is can be found here. We are using v3.0.3. A more easily browsable version is on this site. Finally, if you want a friendlier explanation of the specification, go here.

Debugging REST

One of the challenges in this project will be working with REST servers. Being able to quickly figure out if what you're receiving from the server is correct is going to be important. There are all sorts of tools to help with this – paid, free, online, offline, . . .

I like to go with command line tools, and my favourite for this is cURL (`https://curl.se/`). It's available for most platforms and can do far more than we'll use here.

For online options, the *Advanced REST Client* extension for Google Chrome is fairly simple to use. Or maybe `https://hoppscotch.io/`.

Other questions

Why text–based?

If it's going to be a *game*, why not have it be in Unity or Unreal? From a systems design standpoint, that would place emphasis in the wrong place. Rather, formulating the project this way provides a frame for a multi–tier application with a user interface, database, and API, along with integration with other software platforms. Our interest here is in understanding what it takes to a create a *platform* for a given problem domain. A MUD provides a suitable technical challenge and an opportunity for some creative expression.

How does this support a game that does . . . ?

Does the game have a PvP element? What if there is no combat at all? Or an economy of some sort? It should be possible to support all of these scenarios with game engine primitives. This places pressure to define a robust set of primitives. I have some ideas, and teams have an opportunity to debate this, but emphasis in the project should be to get the platform working first, and not spend excessive time on this.