

Spring 5-1-2024

A UI-Enhanced Approach to Generic Web-Based Scheduling

Tyler Hinrichs

University of Connecticut - Storrs, tylernh99@gmail.com

Follow this and additional works at: https://digitalcommons.lib.uconn.edu/srhonors_theses



Part of the [Software Engineering Commons](#)

Recommended Citation

Hinrichs, Tyler, "A UI-Enhanced Approach to Generic Web-Based Scheduling" (2024). *Honors Scholar Theses*. 986.

https://digitalcommons.lib.uconn.edu/srhonors_theses/986

A UI-Enhanced Approach to Generic Web-Based Scheduling

An Honors Thesis Submitted to
the School of Computing, University of Connecticut,
in Partial Fulfillment of the Requirements
for the Bachelor of Science Degree in Computer Science with Honors



Tyler Hinrichs

May 2024

Thesis Advisor: Dr. Wei Wei

Honors Advisor: Dr. Caiwen Ding

Contents

1	Introduction	1
2	Genetic Algorithm	3
2.1	Background	3
2.2	Simple Solution	7
2.3	TA Solution	9
2.3.1	Data Format and Parsing	9
2.3.2	Algorithm Implementation	11
3	Data Storage	14
3.1	Entity Representation	14
3.1.1	Relational Databases	15
3.1.2	First Schema	15
3.1.3	Improved Schema	18
3.1.4	Incorporating Schedule Generations	22
3.2	Generating Demo Data	24
3.3	Updated Algorithm	25
4	API	29
4.1	Flask	30
4.2	Flask-SQLAlchemy	32
5	UI	35
5.1	React.js	35
5.2	ShadCN UI	42

5.2.1	Component Examples	43
5.3	Application Flow and Design	50
5.3.1	Using the Application	50
5.3.2	Example Schedules	59
6	Conclusion and Future Work	61

List of Figures

1	A simple flow diagram of the genetic algorithm [4]	6
2	UConn CSE TA information used in early implementation of genetic algorithm	10
3	The first relational schema devised for the generic scheduling scenario	17
4	Improved relational schema devised for the generic scheduling scenario	22
5	Adding ScheduleGeneration and associated relations to the schema to save the output of the genetic algorithm	23
6	Converting facilitatoreligibleevents to a more functional format for the genetic algorithm	26
7	An example POST request using query parameters and the request body	32
8	An example of a Flask-SQLAlchemy model class used to map the ScheduleInstance relation to Python objects	34
9	Querying a user's ScheduleInstances using Flask-SQLAlchemy	34
10	An example of the hierarchical structure of React components used in a frontend application [20]	37
11	root.jsx code demonstrating the hierarchical structure of Re- act components	37
12	The application's main page, showing sections of the UI that correspond to UI components	38
13	A simple example of a state's lifecycle in React [16]	40

14	Using <code>createBrowserRouter</code> to specify routes and each corresponding UI component in React	41
15	Example 404 error message shown with <code>ErrorPage</code> when attempting to access an endpoint that does not exist	42
16	A <code>Card</code> element containing a <code>Table</code> element to represent structured schedule data	45
17	A <code>Dialog</code> element used to allow the user to edit the details of an event	45
18	An example of a <code>Toast</code> appearing on the screen after the user successfully updates information about an event	46
19	On a desktop format, the schedule page's navigation bar is positioned horizontally	48
20	At smaller screen sizes, the navigation bar is placed in a <code>Sheet</code> , a modified <code>Dialog</code>	48
21	<code>Form</code> uses React Hook Form and zod to create high-quality forms with data validation.	49
22	UI at the <code>/profiles</code> endpoint	51
23	UI at the <code>/schedules/<scheduleId></code> endpoint	53
24	The timeblocks table at the <code>/schedules/<scheduleId>/timeblocks</code> endpoint	55
25	The timeblock generation form at the <code>/schedules/<scheduleId>/timeblocks</code> endpoint	55
26	UI at the <code>/schedules/<scheduleId>/events</code> endpoint	57
27	UI at the <code>/schedules/<scheduleId>/facilitators</code> endpoint	57

28	Dialog UI at the <code>/schedules/<scheduleId>/facilitators</code> endpoint that accounts for <code>facilitatoreligibleevents</code> and <code>facili- tatoruntimeavails</code>	58
29	UI at the <code>/schedules/<scheduleId>/locations</code> endpoint . .	58

Abstract

Administrative scheduling is a key aspect of a wide variety of systems, but despite being a widespread need, it is not a straightforward task. Organizational uniqueness introduces complexity when attempting to use algorithmic methods to automate scheduling, as individual organizations often have their own ways of determining various details and constraints of a schedule. However, in this paper, we assert that there are relevant commonalities that many different schedules fundamentally possess, allowing us to create a generic scheduling application that can be productively used for as many different scenarios as possible. After devising a schema that captures this generic representation, we expand upon previous efforts of using the genetic algorithm for university scheduling tasks [1] to find valid schedules that adhere to the generic schema. Then, we use React.js, Flask, and MySQL to create a user-facing application that allows users to compute valid schedule instances for their use cases.

1 Introduction

Scheduling is essential for the operation of many systems, whether for a university's computer science department, an academic conference, or a summer violin camp. On the surface, these three entities may seem to have completely unrelated administrative scheduling needs, but when looking more closely, there are more commonalities than often considered. These systems all have recurring events, often occurring on a weekly basis, with requirements spanning location, time, room capacity, and more. The task of scheduling events that prioritize the needs of many groups at once is crucial for many systems to function and will continue to be in the future. Ideally, this task is automated, efficient, and as intuitive as possible. This paper will build on previous propositions of using the genetic algorithm to maximize speed and resource preservation in university scheduling systems [1]. Here, the genetic algorithm will be implemented in a broader context, expanding to any generic scheduling scenario where administrative scheduling is pertinent.

The process of constructing this generic scheduling platform is multifaceted and necessitates quality design of the entire stack. On the backend, this requires an implementation of the genetic algorithm that takes as input stored schedule data. We propose a relational design that attempts to capture generic representations of entities in scheduling scenarios, and we store data that adheres to this schema in a MySQL database. Next, we explore constructing an API in Flask to provide methods of retrieving data from the database, and to provide an interface for a frontend application to interact with the genetic algorithm and the data stored in the database. We

will construct an intuitive web user interface using React.js and the modern component library ShadCN UI to create a user-facing application to carry out scheduling tasks. We emphasize how intuitive design is leveraged to create a high quality user experience. We will conclude with mentions of improvements that could be conducted on this project in the future to further improve several aspects.

2 Genetic Algorithm

The logic to compute the schedule given user-defined entities and constraints is based on the concept of the genetic algorithm. We will thoroughly define the algorithm and demonstrate several examples of it in practice before showing how it applies to the generic scheduling scenario and the entities defined here.

2.1 Background

The genetic algorithm is an optimization algorithm based on the concept of natural selection, evaluating a population of potential solutions and choosing the best instances to create subsequent generations, driving towards an optimal solution [7]. There are several key, distinct components to the genetic algorithm to define.

The first component is the initial population; in a scheduling context, this consists of a set of randomly generated schedules. Each takes into account random choices for all factors involved. For a university scheduling system, this means random choices for when each course is scheduled, which professor is teaching each course, the location where the course will be taught, etc. This population is the genetic algorithm's input and will evolve over time, similar to how evolution of a biological group involves an initial population that undergoes evolution.

Next, a fitness function is used to evaluate the strength of an individual instance of the current population. It evaluates a schedule's quality by outputting a numerical quantification of its validity. The fitness function is at

the crux of the algorithm; it is highly customizable and needs to account for every factor involved, assigning an overall penalty score based on violations that make the schedule invalid. Its input is a schedule, and its output is a float fitness score; generally, if the algorithm finds `penalty_count` penalties, the score is calculated as

$$1/(\text{penalty_count})$$

if `penalty_count` is greater than 1, or 1 if `penalty_count` is 0. Next, there is a function for evolving the population. This function sorts the members of the population by fitness score, and selects the “fittest” individuals, who will be selected to create a new generation. Importantly, there is a degree of randomness and mutation that is added throughout this process, in many ways to mimic mutation seen through Darwin’s theory of evolution in nature [5]. Mutation introduces diversity into a given population, increasing the chance that an optimal solution will be found [7]. In a question where optimizing many parameters is important, even if an initial population has many unique instances, it is not guaranteed that the population has the most optimal traits. Adding mutation in small amounts to the population upon evolution ensures that the population is able to evolve in a sufficiently diverse way, making it more likely that traits unseen during the initial population could appear in future generations (while also ensuring that the high quality attributes of the current fittest individuals are mostly maintained).

In a more concrete sense, when instances of a population are chosen for the next generation, their “offspring” are a random combination of their

traits. The “parents” for an individual in a new generation are randomly chosen from a subset of sufficiently fit members of the current population. A generation also has a given population size; the higher the population size, the more resource-intensive and potentially slow each iteration of the algorithm could be, but more combinations could be created, potentially leading to earlier termination of the algorithm. We can also adjust the rate of mutation; mutation works by randomly generating a new schedule and selecting traits to replace the traits of the individual being mutated at a low probability. Since the random individual is not a member of the initial population, it is possible for them to introduce new, unseen traits into the population.

Finally, the algorithm runs until an optimal solution is found, or until a defined number of generations have passed without finding an optimal solution. The maximum number of generations before termination can be increased if the current amount is too few to reliably find an solution, or decreased if optimal solutions are found much sooner. If the algorithm only terminates after all generations have been carried out, running far beyond the generation when valid schedules are found is a waste of time and resources. If the algorithm terminates upon finding the first valid schedule, there is no such concern of resource usage. The parameter for number of generations, alongside the parameters for probability of mutation and generation size, are important to test and tweak over time.

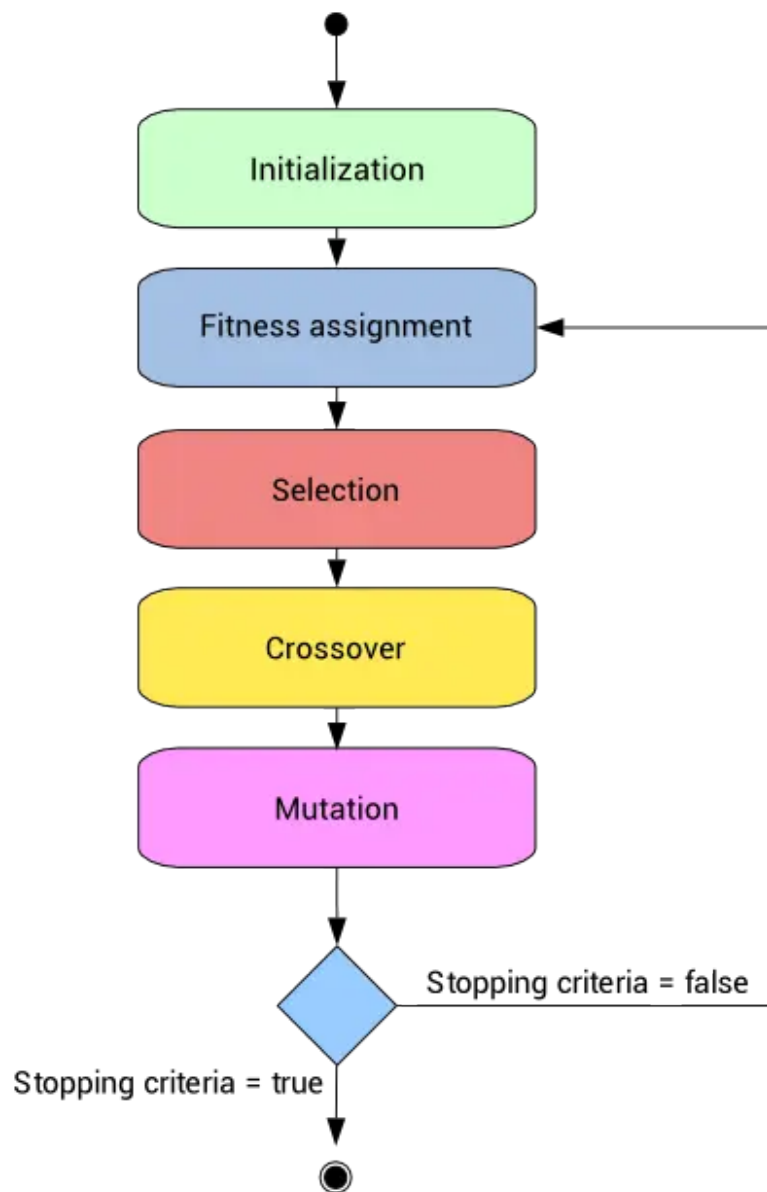


Figure 1: A simple flow diagram of the genetic algorithm [4]

A flow diagram of the genetic algorithm from a high level is shown above in Figure 1. At the top of the diagram, the initialization step occurs. This is where the initial population is derived. This initial population is fed into the next step, fitness assignment, which begins a loop (mirrored with a while loop in the `run` function within the `GeneticAlgorithm` class). From there, the fitness of the current population is calculated. Following fitness assignment, we select the individuals with the highest fitness scores to be part of the population involved with crossover to create the next generation's population. After crossover, we introduce mutation, finally arriving at a sufficiently evolved population. The stopping criteria used in the scheduling problem is finding a schedule with an optimal fitness score; if this has been found, we can exit the loop and terminate the algorithm. Otherwise, the evolved population is sent back to the fitness assignment step and the loop in the algorithm will continue either until a schedule with optimal fitness is found or until the maximum number of generations have occurred, a constant set before the algorithm runs.

2.2 Simple Solution

The first step in the process of representing the genetic algorithm in code was creating a simplified example that was completely independent of the scheduling problem. Given a mathematical function $f(x, y, z) = 5x^3 + 3y^2 + 2z^5 - 25$, the goal was to find values of x , y , and z such that $f(x, y, z)$ evaluated to as close to 0 as possible [19]. The fitness score of the set of variables x, y, z was calculated from first evaluating $f(x, y, z)$, ; if the value

was 0, we could terminate the algorithm, and otherwise, we would calculate

$$\text{fitness_score} = \frac{1}{|f(x, y, z)|}$$

The closer $f(x, y, z)$ evaluates to 0, the smaller the denominator is, and therefore the larger the fitness score is. If `fitness_score` is 100,000 or greater, the algorithm terminates.

The algorithm uses a population size of 1000 and a maximum generations value of 1000. For initialization, 1000 combinations of x , y , and z are generated, creating the initial population. They are evaluated based on the fitness score mentioned above, and sorted based on this scoring. Following this, the top 100 combinations are selected to be part of crossover. Crossover is straightforward; simply add all values of x , y , and z to a set, from which 1000 new combinations are made from randomly selecting three values for x , y , and z . Mutation is introduced by randomly multiplying each variable by a number between 0.99 and 1.01, chosen at an even probability. This outputs an evolved population of combinations, and the process repeats until either 1000 total generations have been evolved or until an ideal combination has been found.

This solution leveraged Python to create a simple yet powerful instance of a functional genetic algorithm. It incorporated all previously mentioned techniques, including generating an initial population, creating a fitness function that evaluated the fitness of an individual instance of the population, and selecting the best solutions from those to create the subsequent generations while applying crossover and mutation.

2.3 TA Solution

The next solution was a simplified version of the scheduling problem that held several variables constant. With these constants, it was possible to use the genetic algorithm for scheduling, while removing some of the complexity of the overall question. This meant the algorithm could be structured used to solve a realistic problem while abstracting away some of the details, reducing some of its complexity.

The solution here revolved around teaching assistants (TAs) for the UConn course CSE 1010. This is a large course that most engineering majors take, with many lab sections and a TA needed for each. The algorithm is used to match TAs to lab sections, making sure each TA led a sufficient number of sections (not too many or too few) and making sure that each section had a TA assigned.

2.3.1 Data Format and Parsing

Data for this problem was given in a CSV format. The first column was labeled “Name”, and had general information about each lab section. For example, the first row had a value of “001L We 8:00am-9:50am” for this attribute, implying that this was lab section 001L, and it occurred on Wednesdays from 8am-9:50am. All following columns represented a specific TA (by identifying their name), and each value in each column was either an X/x or a null value. Any empty cell indicated that the given TA was not available for the given section, whereas an X/x (any non-null value could be treated equivalently) represented a TA’s availability for the corresponding section.

Name	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
001L We 8:00am-9:50am	x		x	X		x		x						x				x		x			X
002L We 10:00am-11:50am	x						x							x	x								
003L We 3:15pm-5:05pm		x			X				x		x	x								x			X
004L We 5:15pm-7:05pm					x	X			x	X	x	x								x			X
005L We 5:15pm-7:05pm					x	X			x	X	x	x									x		X
006L We 9:15am-11:05am	x				x	X								x					x				
007L We 11:15am-1:05pm	x		x					x								x							
011L We 1:15pm-3:05pm	x	x	x			X		x					x				x				x		
012L We 8:00am-9:50am	x			x	x					x									x		x		X
013L Fr 12:00pm-1:50pm			x	x						x												x	x
014L Fr 2:30pm-4:20pm			x							x											x		x
015L Fr 10:00am-11:50am			x		x										x			x				x	
016L Fr 10:00am-11:50am			x			x									x			x				x	
017L Fr 12:00pm-1:50pm			x	x						x					x							x	x
018L Fr 2:00pm-3:50pm			x	x						x					x								x
019L Fr 2:30pm-4:20pm			x							x					x						x		x
021L Tu 10:00am-11:50am	x			x						x	x				x				x	X			X
022L Tu 12:00pm-1:50pm	x			x														x		X			
023L Tu 2:00pm-3:50pm				x											x					X	x		
024L Th 10:00am-11:50am	x									x					x					x	X		X
025L Tu 4:00pm-5:50pm							X								x					x			x
026L Th 4:00pm-5:50pm							X								x					x			x
027L Th 2:00pm-3:50pm														X						x			X
028L Th 3:00pm-4:50pm														X						x	x		X
031L Th 10:00am-11:50am	x									x						x				x	X		X
032L Th 10:00am-11:50am	x									x						x				x	X		X
033L Th 12:00pm-1:50pm	x			x						x	x				x					X			X
034L Th 1:00pm-2:50pm										x					X					X	x		X
035L Th 5:00pm-6:30pm	x									x					X	x	x				x		X
036L Th 1:00pm-2:50pm										x					X	x				X	x		X
037L Th 3:00pm-4:50pm															X					x			X
038L Th 5:00pm-6:50pm	x						x			x	X	x	x							x			X
041L Tu 10:00am-11:50am	x			x						x	x									x	X		X
042L Tu 1:00pm-2:50pm										x					x					X	x		
043L Tu 3:00pm-4:50pm										x	x									x			
044L Tu 5:00pm-6:50pm	x	x	x							x										x		x	x
051L Tu 10:00am-11:50am	x			x						x	x									x	X		X
052L Tu 1:00pm-2:50pm	x			x						x					x					X	x		
053L Tu 3:00pm-4:50pm															x	x					x		
054L Tu 5:00pm-6:50pm	x	x	x							X	x				x						x	x	x

Figure 2: UConn CSE TA information used in early implementation of genetic algorithm

In order to parse the data from the CSV, the Python package **pandas** was leveraged to parse information into arrays of information contained in the CSV. These arrays are as follows:

1. **courses:** `list[Course]` - Course objects for each lab section
2. **teaching_assistants:** `list[{int}]` - List of sets of integers corresponding to indices of lab sections for which each TA is available

2.3.2 Algorithm Implementation

In this problem, we begin representing entities as Python classes, starting with two in the TA problem: **Course** and **Schedule**. **Course** represents a generic academic event, meaning that lab sections are represented as **Course** objects. **Course** has a constructor that takes in a string variable called `timeblock_str`, which is parsed into meaningful information regarding the time and days of the week of the section. The **Course** object also has a function `overlap(self, other)` which takes in another **Course** object and checks if the two schedules overlap.

Next, the **Schedule** object holds schedule information and is an instance of a population used in the genetic algorithm. It is initialized with a list of **Course** objects, TA availabilities, and course assignments. Course assignments are an optional parameter that if left blank are generated at random. This is useful when creating the first population of random schedules, and when retrieving a random schedule to use for mutation. TA availabilities are also passed to **Schedule**, allowing us to assign TAs to sections they can be present for. Each **Schedule** maintains a list of **Course** objects where each

index corresponds to the same **Course** to be scheduled. This makes it easy to compare **Course** information across different **Schedules**.

The fitness score is computed in **Schedule** by finding the number of penalties in the current **Schedule** object's assignments, returning 0 if there are no penalties or $(1/penalty_count)$ otherwise. Penalties are deducted if a TA has too many or too few assignments, or if they are assigned overlapping sections. The fitness score is lazily loaded as it is only calculated the first time `getFitness(self)` is called, after which it can be fetched from a variable.

The **GeneticAlgorithm** class incorporates **Course** objects, **Schedule** objects, and information about TAs and attempts to find valid schedules. We use the following values:

- Maximum number of generations: `MAX_GEN = 1000`
- Population size: `POP_SIZE = 100`
- Amount of mutation: `MUTATION_AMT = 0.05`

The **GeneticAlgorithm** class takes as input a list of **Course** objects and a list of TA availabilities as defined previously. It has the following functions defined:

- `evolve(self)`
- `crossover_two_schedules(self, a: Schedule, b: Schedule, rate_for_a = 0.5)`
- `mutate_schedule(self, mutation_schedule: schedule, schedule: Schedule)`

- `run(self)`

When `run` is called, `POP_SIZE` random schedules are created `Schedule` objects without passing in an object for course assignments. The `run(self)` function is what initiates the `GeneticAlgorithm`. It first sorts the current list of schedules; then, while the number of total generations is below `MAX_GEN` and the most optimal schedule has a fitness score less than 1, it calls `sort` on the current population, then `evolve(self)` is called, and the total number of generations is incremented. After breaking out of the while loop, there is a check whether the ending population's best schedule had a fitness score of 1, and it outputs the score.

`evolve` finds the top 20% of schedules based on fitness score and uses `crossover_two_schedules` to create the new generation based on random pairs from that population. Each `Schedule` is sent through the `mutate_schedule` function, which uses `crossover_two_schedules` on the original schedule and a randomly generated mutation schedule with an uneven probability rate. `MUTATION_AMT` is set to 0.05, meaning that traits from the mutated schedule are chosen only 5% of the time; the mutated schedule does not have a huge effect on the population, but can add some diversity.

In a test of 100 trials, a viable solution (one with a perfect fitness score) was found every time, with an average time of 0.5867 seconds being taken for each run.

3 Data Storage

A crucial step in the process creating a generic scheduling system is finding how to represent data in a general way. The TA solution demonstrates usage of the genetic algorithm, but it oversimplifies the schema needed for a sufficiently robust and generic solution. We will expand upon the TA solution to include additional entities that need representations in generic scheduling scenarios. Each entity can exist as a relation in a relational database, which can then be mapped to classes in code that can be used in the genetic algorithm.

Once a user uploads or enters data for their schedule, this data must be stored. User experience is inherently tied to long-term usability, a major part of which is maintaining a user profile, and keeping track of user data and activity so they can return to a saved state in the future. In full stack applications, data storage through some form of a database is a necessity. The backend code that runs the genetic algorithm needs to access data from the database, and needs to be able to convert representations from the database to representations in code. In this section, we will discuss how the entities of the schema we create are best represented in a database, and subsequently, ways in which they will be represented in code.

3.1 Entity Representation

Before interacting with a database or making classes to use in the genetic algorithm, we must find what we need to represent in our schema, and how precisely we want to represent them. Here, we will explore several schema

approaches, each building on the previous, to get to a final schema that can be used in a relational database like MySQL.

3.1.1 Relational Databases

A relational database approach was used for data storage. Relational databases are known for their strength in representing highly structured data with strict, expected formats for each entity [3]. This matches the current use case; in an attempt to promote generality, entities should have baseline, unchangeable characteristics that all, or at least most, scheduling scenarios would have. Relational databases represent entities as relations within a higher schema, with relationships imposed on relations themselves (such as datatype for a particular attribute), or relationships between relations (such as a foreign key relationship from an attribute of one relation to primary key attribute).

3.1.2 First Schema

The first schema representation of the problem has relations for several main entities in the problem. First, we have **Facilitator**; this represents a person that can lead or facilitate an event. This generic representation is meant to take the place of anything from a professor for a university course to a counselor at a summer camp. Next, we will use **RecurringEvent** to represent an event that occurs on a recurring basis. **Location** is used to generically represent a place where a **RecurringEvent** could be scheduled to occur. There is a **TimeBlock** to represent the time periods when a **RecurringEvent** can be scheduled, as well as to represent availabilities of **Facilitators**. There are

two further relations to represent the events that **Facilitators** are eligible to facilitate and the **TimeBlocks** that each **Facilitator** is available for, called **FacilitatorEligibleEvents** and **FacilitatorTimeAvails** respectively.

The database structures the data such that there are **Facilitators**, **RecurringEvents** that the **Facilitators** oversee, and **TimeBlocks** and **Locations** when and where the events can occur respectively. Adding in availabilities and eligibilities of **Facilitators**, we get a working schema that is much more detailed than the TA problem, and encapsulates much of the fundamental nature of the question. The schema is shown below in Figure 3, with each relation being represented with its attributes alongside their corresponding datatypes. The primary keys of each relation are bolded.

Initial Scheduling Schema (PKs bolded):

Facilitator

- **facilitatorID**: Integer
- First: varchar(64)
- Last: varchar(64)

RecurringEvent

- **recEventID**: Integer
- eventName: varchar(128)
- maxSize: Integer

Location

- **locationID**: Integer
- locationName: varchar(128)
- locationNumber: Integer
- maxCapacity: Integer

FacilitatorEligibleEvents

- **facilitatorID**: Integer (FK to Facilitator.facilitatorId)
- **recEventID**: Integer (FK to RecurringEvent.recEventId)

FacilitatorTimeAvails

- **facilitatorID**: Integer (FK to Facilitator.facilitatorId)
- **timeBlockID**: Integer (FK to TimeBlock.timeBlockID)

TimeBlock

- **timeBlockID**: Integer
- MWF: Boolean
- start: Time
- end: Time

Figure 3: The first relational schema devised for the generic scheduling scenario

The schema defines more granular relationships between the entities than previously described; for instance, foreign keys are listed, showing which attributes are bound to the set of values of a primary key of some relation. Additionally, there are several further nuances represented here; for example, the `Location` relation has a `maxCapacity` integer attribute, and the `RecurringEvent` has a `maxSize` integer attribute. These can be used to prioritize scheduling `RecurringEvents` in `Locations` that have more, but not much more, capacity than the event's maximum size. It will also allow us to ensure that a `RecurringEvent` is only scheduled for a `Location` that can support its potential attendance size.

3.1.3 Improved Schema

The current design devised to this point only captures the information of a singular schedule; however, the application being built is intended to be used by multiple users, each of which can have multiple schedules that are independent of one another. With this in mind, we add two relations, `User` and `ScheduleInstance`. `User` stores data of a person using the application, including some simple fields such as name, username, etc. `ScheduleInstance` represents a schedule that a user can create and add data to. With this in mind, a `User` can have multiple `ScheduleInstances`, and each `ScheduleInstance` can have multiple `RecurringEvents`, `Facilitators`, etc. There are foreign key relationships that tie these entities together. Using foreign keys ensures that the data stays in first normal form; instead of having multi-valued attributes in a relation, we split such a relation into multiple relations, ensuring atomicity and avoiding heavy coupling of separate rela-

tions [6].

In the new design, there are several changes made to relations in the current schema. Attributes of several relations have been adjusted to capture more characteristics of the problem at hand. Notably, the original schema's version of **TimeBlock** included an attribute called **MWF**, which was a boolean meant to represent whether the event was to occur on Monday, Wednesday, and Friday, or Tuesday and Thursday. This concept was too restrictive and greatly reduced schedule flexibility. **TimeBlocks** were changed to have the attributes **dayOfWeek**, **start**, and **end**, giving users more flexibility to customize timing in their schedule. However, as will be discussed further in later sections, guidelines were added to forms in **TimeBlock** creation that make them standardized in length of time and time between. Each **TimeBlock** is considered an independent unit that occurs on a particular day, instead of being tied to a predefined subset of the days of the week. Other changes include adding **numBlocks** to **RecurringEvent**, which allows for events to be scheduled across one or more events throughout the week. The adjusted schema is shown in Figure 4.

Improved Scheduling Schema (PKs bolded, FKs italicized)

User

- **userID**: varchar(36) – UUID
- username: varchar(32) NOT NULL UNIQUE
- firstName: varchar(128)
- lastName: varchar(128)
- email: nvarchar(256)
- dateCreated: timestamp NOT NULL (default: CURRENT_TIMESTAMP)

ScheduleInstance

- **scheduleInstanceID**: varchar(36) – UUID
- userID: varchar(36) – UUID (FK to User.userID)
- scheduleName: varchar(128)
- dateCreated: timestamp NOT NULL (default: CURRENT_TIMESTAMP)
- lastUpdated: timestamp NOT NULL (default: CURRENT_TIMESTAMP)
- description: Text

Facilitator

- **facilitatorID**: Integer AUTO INCREMENT
- scheduleInstanceID: varchar(36) – UUID (FK to ScheduleInstance.scheduleInstanceID)
- firstName: varchar(128)
- lastName: varchar(128)
- title: varchar(128)
- bio: TEXT

RecurringEvent

- **recEventID**: Integer
- scheduleInstanceID: FK to ScheduleInstance.scheduleInstanceID
- eventName: varchar(128)
- maxAttendance: Integer
- numBlocks: Integer

Location

- **locationID**: Integer
- scheduleInstanceID: FK to ScheduleInstance.scheduleInstanceID
- locationName: varchar(128)
- locationNumber: Integer
- maxCapacity: Integer

FacilitatorEligibleEvents

- **facilitatorID**: Integer (FK to Facilitator.facilitatorId)
- **recEventID**: Integer (FK to RecurringEvent.recEventId)

FacilitatorTimeAvails

- **facilitatorID**: Integer (FK to Facilitator.facilitatorId)
- **timeBlockID**: Integer (FK to TimeBlock.timeBlockID)

TimeBlock

- **timeBlockID**: Integer
- **scheduleInstanceID**: FK to ScheduleInstance.scheduleInstanceID
- **dayOfWeek**: ENUM('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
- **start**: Time
- **end**: Time

Figure 4: Improved relational schema devised for the generic scheduling scenario

3.1.4 Incorporating Schedule Generations

This schema now sufficiently captures a baseline amount of information for users and schedules in our applications. However, the entities captured in the schema so far is only the input data for the genetic algorithm. Once the algorithm finishes processing, it outputs information for each **RecurringEvent**, defining how each should be scheduled for a valid **ScheduleInstance**; this includes **Facilitators**, **TimeBlocks**, and **Locations** associated with each event. In a complex scheduling scenario, there could be numerous different ways of devising how these events should occur. In the application itself, users can run the algorithm as much as they desire to compute a new generation of the **ScheduleInstance**. The relation **ScheduleGeneration** is introduced, a scheduled version of **ScheduleInstance**. To adhere to first normal form, we further introduce the relation **ScheduledEvent**, **ScheduledEventLocation**, **ScheduledEventFacilitator**, and **ScheduledEventTimeblock**. These relations will store the output from the genetic algorithm and will allow a user to access the saved result of how their **ScheduleInstance** could validly occur.

Schedule Generation Schema (PKs bolded)

ScheduleGeneration

- **scheduleGenerationId**: varchar(36) – UUID
- name: varchar(128)
- description: TEXT
- scheduleInstanceId: varchar(36) – UUID (FK to ScheduleInstance.scheduleInstanceId)

ScheduledEvent

- **scheduledEventId**: varchar(36) – UUID
- name: VARCHAR(128)
- scheduleGenerationId: varchar(36) – UUID (FK to ScheduleGeneration.scheduleGenerationId)
- recurringEventId: Integer (FK to RecurringEvent.recEventID)

ScheduledEventLocation

- scheduledEventId: varchar(36) – UUID (FK to ScheduledEvent.scheduledEventId)
- locationID: Integer (FK to Location.locationID)

ScheduledEventFacilitator

- scheduledEventId: varchar(36) – UUID (FK to ScheduledEvent.scheduledEventId)
- facilitatorId: Integer (FK to Facilitator.facilitatorID)

ScheduledEventTimeblock

- scheduledEventId: varchar(36) – UUID (FK to ScheduledEvent.scheduledEventId)
- timeBlockID: Integer (FK to TimeBlock.timeBlockID)

Figure 5: Adding **ScheduleGeneration** and associated relations to the schema to save the output of the genetic algorithm

3.2 Generating Demo Data

Whereas the TA problem had concrete data already prepared, there was no preexisting data readily available for the new schema; as such, data was created in order to build an early proof of concept, especially prior to the creation of a frontend application that would allow a user to easily create and modify data.

A script was created to generate **pandas** dataframes for each relation mentioned in the schema above. Data generation included using the **names** pip package provided by Python to get a list of 20 facilitators, and then using the **random** module to choose which eligibilities and availabilities they had for **RecurringEvents** and **TimeBlocks** respectively. Data was assigned to entities in attempt to simulate a real-world situation; for example, in a university scheduling system, we can assume that **Facilitators** will have a range of eligibilities, while a university is likely to have hired at least one professor for each course. Otherwise, valid schedules would be impossible.

As mentioned earlier, there are some more nuances that we may want to consider in the future. For example, while we have added support for **Facilitator** time availabilities and event eligibilities, we do not currently keep track of preferences that **Facilitators** might have for either. This was an aspect of previous research on this topic [1], and while it was not considered in this paper, it is suggested to add back to the schema in future implementations.

The generated data was then sent to CSV files using the **to_csv** API provided by **pandas**. This was the first step in getting the data to a MySQL in-

stance. Such an instance was created and connected to through Python code in a script called `table_creation.py`. There, Python code accessed and ran code from a SQL script called `table_creation.sql` in order to clear the current schema and add new, empty relations. Then, after navigating to the MySQL Workbench user interface, the `table_population.sql` script was run, which populated the MySQL with the previously generated data. Later, it would be possible to add data through the frontend that was created. However, as API development preceded frontend development, automating the process of adding valid data to the database greatly quickened the process.

3.3 Updated Algorithm

The updated algorithm must incorporate entities of the new schema and find valid schedules based on the data they specify. Therefore, the algorithm must receive relevant `ScheduleInstance` data as input that it can process and produce a valid `ScheduleGeneration` as output (if it is possible to generate a schedule given the user-specified constraints). At this point, we have only introduced the concept the database, and we have not yet talked about retrieving data from the database and using it in our code. On a low level, this will be completed using the ORM Flask-SQLAlchemy, something that will be discussed in more detail in later sections. However, for a description of the new algorithm, it is sufficient to know that our data will be retrieved from the MySQL database and placed into objects of classes that correspond to relations in our schema.

The structure of the genetic algorithm remains largely the same across

different domains, but there are several areas that must be customized based on the current use case. Therefore, it was possible to use a similar high-level structure to the TA solution found earlier, but several major changes were made. Firstly, the inputs to the classes changes. Now, `GeneticAlgorithm` and `Schedule` take in several more lists: `recurringevents`, `timeblocks`, `facilitators`, `locations`, `facilitatoreligibleevents`, and `facilitatortimeavails`. Each of these are lists of objects that correspond to their respective names (i.e., `recurringevents` is a list of `RecurringEvent` objects). The only exception is that `facilitatoreligibleevents` was converted into a list of tuples to make certain operations simpler when calculating `Schedule` fitness.

```
self.facilitatoreligibleevents = [(fe.facilitatorID,
fe.recurringEventID) for fe in facilitatoreligibleevents]
```

Figure 6: Converting `facilitatoreligibleevents` to a more functional format for the genetic algorithm

Logic for `evolve`, `mutate_schedule`, and `run` largely stay the same, but functionality for `crossoverTwoSchedules` within `GeneticAlgorithm` and `calcFitness` in the `Schedule` class were significantly changed. Furthermore, a simple internal class called `ScheduledEvent` was created to deal with versions of scheduled events internally in the algorithm; they would later be mapped to the ORM's version of `ScheduledEvent` once the algorithm found a valid solution.

`crossoverTwoSchedules` was changed to introduce more complexity when two schedules were combined. Previously, each `Schedule` in the algorithm

had a list of events, where each corresponded to each other. A new **Schedule** made from combining two **Schedules** would be made by randomly selecting the event from either the first or second schedule. However, now that each event has more nuance, a new strategy was introduced. Like before, the **rate_for_a** parameter defines the likelihood of selecting an event from **Schedule A**. If left to the default of 0.5, the probability of selecting from each **Schedule** is equal. If the value for A is closer to 1, then the probability becomes more skewed towards selecting information from A, which can be useful in a mutation scenario where we want information mostly from one **Schedule** or the other.

We now introduce the concept of mixing vs. choosing within **crossoverTwoSchedules**. Choosing is identical to how we previously crossed over two **Schedules**; an event is chosen from either the first or second **Schedule**. Mixing is the concept of selecting subcomponents of each of the events of the two **Schedules**. For example, we might take the **locations** of the first **Schedule**, but the **Facilitators** of the second **Schedule**. Now, **rate_for_a** has two uses; it is used to determine whether we will mix or choose, and then once selected, it defines the probability that elements will be taken from the first or second **Schedule**. The motivation here is that we can introduce more variation when combining **Schedules**, hopefully avoiding the risk of not being able to find any valid **Schedules** due to the necessary arrangement of information not being present.

calcFitness was changed to support the greater complexity brought about by the new schema. Now, penalties are applied if the following are not met:

- Each `Facilitator` is eligible for the `RecurringEvent`
- Each `Facilitator` is available for the `TimeBlock`
 - They specified they are available at this time
 - They are not already assigned to another `RecurringEvent` during this time
- The `Location` has enough capacity for the `RecurringEvent`
- The `Location` is available capacity during the `TimeBlock`

Penalties are similarly assigned as before if violations to these conditions are made. The overall fitness score for a `Schedule` is calculated and used to compare `Schedule` viability.

The `GeneticAlgorithm` class itself is accessible to the Flask API (more information on this in the next section), so it is straightforward to send inputs to the algorithm and

4 API

A crucial aspect of software applications is the flow of data from a data storage mechanism to and from a frontend application that a user interacts with. Such communication has become standardized through REST APIs. An API allows two separate applications to communicate with one another; APIs over HTTP are considered RESTful if they adhere to Representational State Transfer guidelines, meaning that they return a standardized representation of a resource such as JSON, there are expected response formats, and communication is stateless, amongst other REST principles [14]. REST is considered a best practice for professional-grade applications, and our APIs will adhere to these guidelines as data flows from the frontend to the MySQL database and back.

We chose Flask as our framework to create REST APIs for several reasons. First, the genetic algorithm was being created in Python, making it easy to interface with Flask and various methods we create in a Flask-based API. Furthermore, Flask is a lightweight application, and it allows for rapid development, making it apt for our current proof-of-concept use case. The authors' familiarity with the framework also contributed to it being chosen for the MVP, but FastAPI also stood out as a high-quality Python framework that met the project's needs; it is an option worth researching in future iterations of the application to find how it would stack up in terms of efficiency and other metrics when compared to Flask.

4.1 Flask

The Flask framework was suitable for the needs of the application's API. The proof-of-concept consisted of locally running services, and no deployment occurred in the duration of this project; therefore, the Flask API ran on localhost:8080. All API endpoints were in the format `/api/<endpoint>`. There were also several test endpoints for checking the functionality of the genetic algorithm and related classes; these endpoints were in the format `/test/<endpoint>`. Test endpoints were solely used for testing purposes and were not used by the application whatsoever. However, the other endpoints were used in various places in the React.js frontend.

Four HTTP methods were used in the API: GET, PUT, POST, and DELETE. GET is used to retrieve data from the database, and was often used either when a component loaded, or after an update was made to records in the database that required fetching data to keep the UI in sync with data storage. POST was used for users to create new entities in the database, such as adding facilitators, events, and more. PUT was used to edit these entities after creation, whereas DELETE was used to remove these entities. Having these methods available allowed the frontend application to provide users with full control over the data of their schedules.

Earlier, the **User** relation was mentioned and added to the schema; while present in the schema, it has not been fully implemented. As the focus of this application was on the scheduling aspect of the problem, no authentication or login system was made for different users to use the application. Instead, a default user was generated.; its `userID` was saved in the `.env` file in the base

directory, and it was fetched by the Flask API to simulate the current `user` being "logged in". This was sufficient when developing our proof-of-concept.

Next, a standardized JSON response was used as a response for all endpoints. It included a `"statusCode"`, a `"message"`, and a `"body"`, where the `"body"` would hold other information needed. Appropriate error codes were used, such as 200 for general success, 204 for successful deletion, and 404 for an item not being found. If a known error was found and caught, then a specific error message and status code were sent back to the client. However, if the error was not known, a 500 error would be sent back (indicating a general server error).

If an error occurred in the application, a function called `recover_after_error` dealt with handling what the API was to do when an error was encountered. It first used a logger from the `logging` Python module to describe the error and write it to a file called `app.log`. Then, the error was printed to give a verbose description to someone monitoring the API. Finally, the database session was rolled back. In the next section, more information will be given on database interactions such as this; however, it is worth noting here that the rollback ensured that if an error happened at any point during the process of an API call, those changes were undone to ensure that no partially-successful changes were allowed in the database, which would cause unexpected behavior and malformed data.

The APIs themselves used Flask routes, each of which corresponds to a function that is run whenever the endpoint specified by the route is hit. Some data was passed through both the query parameters of the url, whereas some data was passed in the of the request. Both methods were able

to be consumed by the functions themselves, and data could be accessed and used to interact with the database. The general paradigm followed was if there was an endpoint that needed to access a specific instance of an entity, or a subset of entities, the identifier was included in the query parameters, whereas further information was included in the body.

For example, the following endpoint was used for a user to POST timeblock information based on the current `ScheduleInstance` they were interacting with. The specific `ScheduleInstanceId` was passed as a query parameter which is mapped to a parameter of the route's Python function; further information about how to generate timeblocks was sent via the body of the request, and fetched from the `request.json` map.

```
@app.route('/api/schedules/<scheduleInstanceId>/timeblocks',
methods=["POST"])
def add_timeblocks(scheduleInstanceId):
    ...
    data = request.json
    numBlocks = data.get("numBlocks")
    blockLength = data.get("blockLength")
    timeBetween = data.get("timeBetween")
```

Figure 7: An example POST request using query parameters and the request body

4.2 Flask-SQLAlchemy

The process of getting data from a relational format, such as being stored in our MySQL database, to a usable format that can be processed by the API and genetic algorithm, is a nontrivial task. It is possible for us to set

up Python classes from scratch and handle all error checking, data type checks, and more details; however, Object-Relational Mappings (ORMs) are libraries that can abstract away some mapping details, giving the developers a standardized way of transferring their data from a relational database to a Python class, and vice versa [2]. ORM works towards minimizing the complexity added due to impedance mismatch, a common issue where data that shows up in an object in code might not be able to be captured in a single relation [18]. A good example of this is a class `ScheduledEvent` which was used in conjunction with `GeneticAlgorithm`; this class has reference to an event, and collections of facilitators, locations, and timeblocks. For our data to be in first normal form, we cannot have all of this data in one relation, but we can use an ORM to help us split this data into the four corresponding relations and store data as such in the database (`ScheduledEvent`, `ScheduledEventFacilitator`, `ScheduledEventLocation`, and `ScheduledEventTimeblock`).

Here, we use Flask-SQLAlchemy; this ORM is built on top of SQLAlchemy, an ORM which maps data from SQL databases into Python classes. Flask-SQLAlchemy builds on the functionality of SQLAlchemy and provides several useful extensions for Flask applications. We set up a Python module called `models.py` which contains all classes to be mapped to and from our MySQL database. Every class has variables for each of the attributes of the table, with specific information on the data type, whether the attribute is a primary key, and other relationships such as if it is a foreign key. Then, in `app.py`, the module where our API code is held, we initialize our Flask app to access the local MySQL instance. Now, we can use the classes in `models.py` to

ensure that our data going to and from the database is in the proper format, and our genetic algorithm can use instances of these classes directly.

An ORM is also useful when wanting to ensure that we have data integrity; we can set certain datatypes such as an integer or constant length string, which can be done in the ORM by specifying `db.Integer` and `db.String(36)` respectively (the latter is a string of length 36, a length used by UUIDs).

```
class ScheduleInstance(db.Model):
    __tablename__ = "scheduleinstance"
    scheduleInstanceID = db.Column(db.String(36), primary_key=True)
    ...
    description = db.Column(db.Text)
```

Figure 8: An example of a Flask-SQLAlchemy model class used to map the `ScheduleInstance` relation to Python objects

Using Flask-SQLAlchemy in code can be completed by calling methods on model classes themselves, and using querying and filtering techniques to retrieve specific subsets of data.

```
schedules = ScheduleInstance.query.filter_by(userID=user_id).all()
```

Figure 9: Querying a user's `ScheduleInstances` using Flask-SQLAlchemy

5 UI

The user interface (UI) and user experience (UX) were prioritized aspects of this application. A high-quality interface must be built in an intuitive way that is understandable and allows users to have a seamless experience while interacting with its functionality. It must balance simplicity and usability; we want to give users the ability to complete potentially complex tasks in our application while reducing clutter and avoiding a convoluted experience.

5.1 React.js

React.js has been an industry standard for building high-quality frontend applications in recent years. It is a JavaScript library made for modular user interfaces in the form of UI components [8]. Components are discrete units of UI code packaged together, allowing them to be reused throughout an application. Components themselves are functions written with JSX, a markup syntax that synthesizes HTML, JavaScript, and CSS [8]. Components can be nested within one another, allowing for pages and applications to be composed of multiple components. Figure 10 provides an example hierarchy of how an **App** component could be composed of multiple nested elements. In our application’s **Root** element contained within `root.jsx`, we have several nested components; **Header**, **Footer**, **Toaster**, and **Outlet**. **Outlet** is used as a placeholder to render other components, and will be used to render the main UI elements on the page. The division of UI components often can correspond to visually and intuitively separate aspects of the application. For example, in the main page of our app, there is a navigation bar, a

footer, a card with help information, and a card displaying the user's schedules. As expected, each of these are individual components. Both the help information card and the schedule card are children of a component called `HomeContent` which also includes the UI element greeting the user as the result of an API call fetching user data. This visual is shown in Figure 12. React allows us to make our UI modular, which is useful for organization and code reuse. Components allow us to section logical sections of code into their own independent units that can be reused throughout our application.

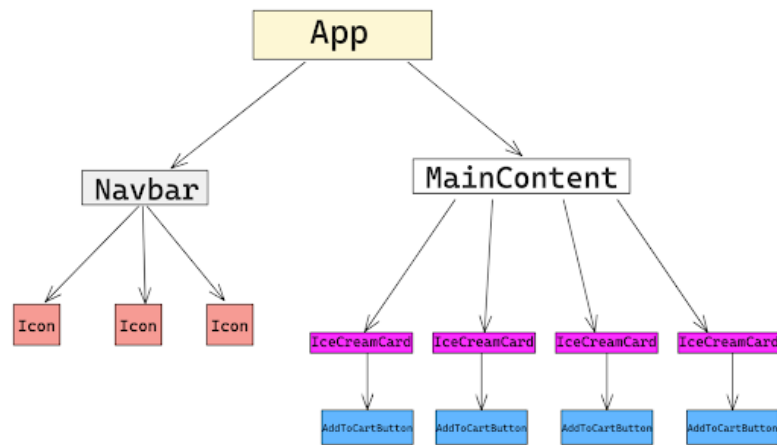


Figure 10: An example of the hierarchical structure of React components used in a frontend application [20]

```
export default function Root() {
  return (
    <>
      <div className="flex flex-col min-h-screen">
        <Header />
        <div className='nav-space'></div>
        <div id="content">
          <Outlet />
        </div>
        <Footer />
      </div>
      <Toaster />
    </>
  );
}
```

Figure 11: `root.jsx` code demonstrating the hierarchical structure of React components

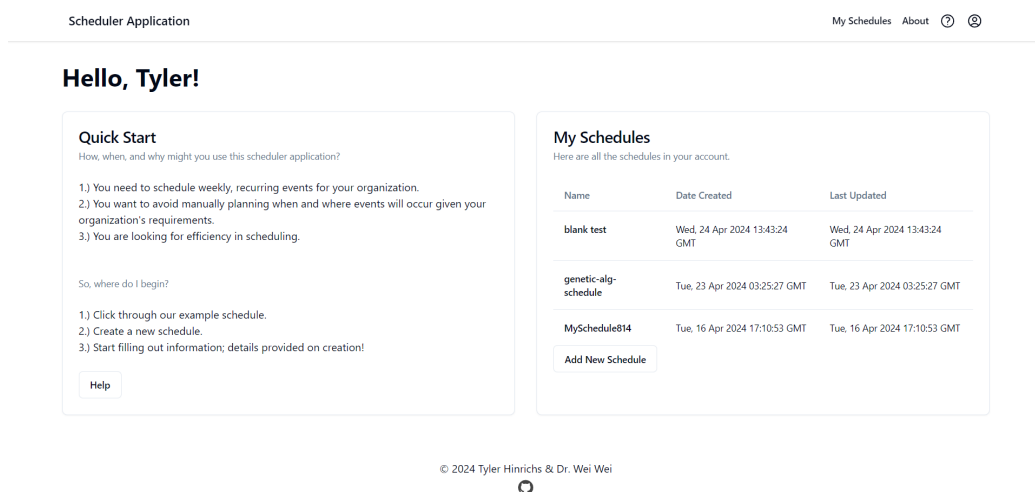


Figure 12: The application's main page, showing sections of the UI that correspond to UI components

Components maintain data in the form of states. First, data can be passed to a React component through props, key-value parameters of a component through which parent components can pass objects to child components. The `useState` Hook (a Hook is a type of React function) is used to set up a state, initializing a variable that stores the contents of the state and a custom setter used to update that variable [8]. The `useEffect` Hook is used to trigger code to run when certain elements in the component have changed. `useEffect` can be triggered as a result of state or props changing, and within a `useEffect`, state setters can be used to update certain elements based on the result of a change within the component. Figure 13 provides an example of the lifecycle of a state.

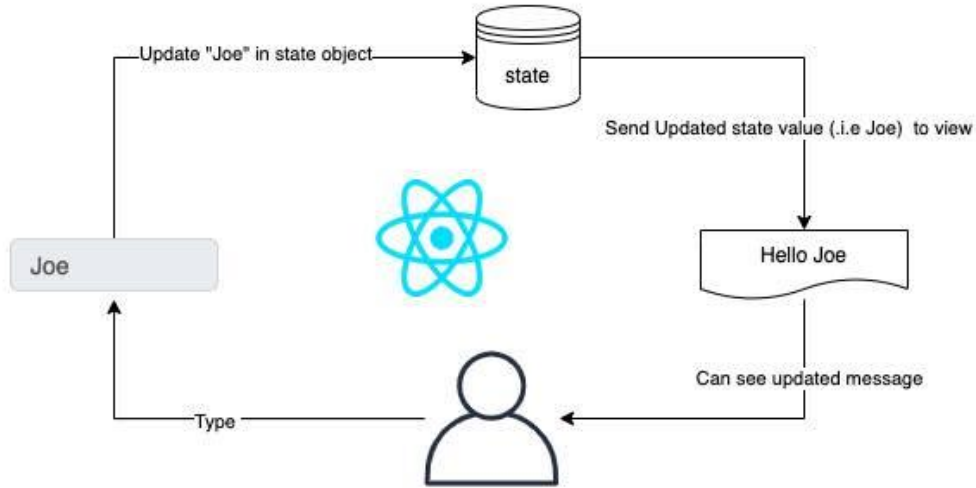


Figure 13: A simple example of a state's lifecycle in React [16]

In addition to these baseline features of React, we used React Router, a library that allows for client side routing within a React application [13]. This allows us to create URL paths that the user can navigate between without initiating page reloads; instead of fetching new content from the server, the page renders UI elements on the client [13]. Within `main.jsx` in our application, we use `createBrowserRouter` to specify the UI elements to be shown on each endpoint.


```

const router = createBrowserRouter([
  {
    path: "/",
    element: <Root/>,
    errorElement: <ErrorPage/>,
    children: [
      {
        path: "/",
        element: <HomeContent/>,
      },
      {
        path: "schedules/",
        element: <SchedulesPage/>,
      },
    ],
  },
]);

```

Figure 14: Using `createBrowserRouter` to specify routes and each corresponding UI component in React

We can also define an `errorElement` that shows a specific UI component when an error is encountered. In our case, we define `ErrorPage` within `error-page.jsx` that receives the error using the React Router function `useRouteError` and displays information inside of a custom error component we created called `ErrorMessage`.

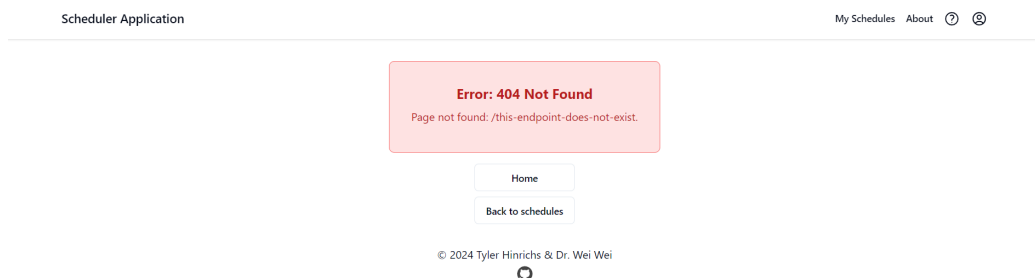


Figure 15: Example 404 error message shown with `ErrorPage` when attempting to access an endpoint that does not exist

If dynamic retrieval of new data is necessary, API requests can be made on an as-needed basis. API calls made over HTTP in React can be made in several ways. The primary method we use is through the axios library, which gives us a reliable way of making HTTP requests to our backend methods running in Flask. While the fetch API can natively be used to make HTTP requests in React, we chose axios as it has better support across multiple browser types and ages, and it has increased capabilities beyond those of fetch including automatic parsing of responses into JSON [12].

5.2 ShadCN UI

ShadCN UI is an open-source collection of modern React components; it is not a traditional UI library as it cannot be installed in a React application with npm (Node package manager), and instead, developers can copy and paste the code for various UI components directly into their codebase, op-

tionally modifying their functionality and styling [15]. The components are styled through Tailwind CSS; their standardized appearance is cohesive and sleek out of the box.

ShadCN has support for numerous types of standard UI components used in modern frontend applications. Examples include dialogs, inputs, tables, checkboxes, and more. ShadCN conveniently allows for CLI commands to copy component code directly into a local directory, where it traditionally sits in the `src\components\ui` directory. There, the code can be modified to fit the application's specific use case. Manually downloading the code when needed allows for a more lightweight application that only contains necessary resources.

5.2.1 Component Examples

Here, we will go over several examples of ShadCN UI components in our application. This list is not exhaustive and instead is meant to describe a few examples of their usage in the context of the scheduling scenario.

One component design that was used at several points in our application was a combination of `Card` and `Table` to represent lists of entities fetched from the database. The `Card` structure allowed for the content to be placed visually into its own section, whereas the `Table` provided a seamless way of showing structured data. This visual is shown in Figure 16 below.

Another ShadCN UI element that was frequently used in the application was `Dialog`. When or deleting an item, a `Dialog` was used to give the user a place to input data or select a button to confirm, submit, or cancel an action. Further UI elements could be placed inside the `Dialog` to carry out

this functionality. An example of using a **Dialog** to edit the details of an event is shown in Figure 17 below.

When a user attempted to complete an action that interacted with the database, a **Toast** was used to show the user a temporary message indicating whether the action was successful or not. The user is able to clear the **Toast** manually, or without action, the **Toast** will disappear on its own. An example of a **Toast** used to indicate that a user successfully updated the details of an event is shown below in Figure 18.

My Schedules

Here are all the schedules in your account.

Name	Date Created	Last Updated
blank test	Wed, 24 Apr 2024 13:43:24 GMT	Wed, 24 Apr 2024 13:43:24 GMT
genetic-alg-schedule	Tue, 23 Apr 2024 03:25:27 GMT	Tue, 23 Apr 2024 03:25:27 GMT
MySchedule814	Tue, 16 Apr 2024 17:10:53 GMT	Tue, 16 Apr 2024 17:10:53 GMT

Add New Schedule

Figure 16: A **Card** element containing a **Table** element to represent structured schedule data

Edit Event

Event Name

Max Attendance

Number of Blocks

Update

Figure 17: A **Dialog** element used to allow the user to edit the details of an event

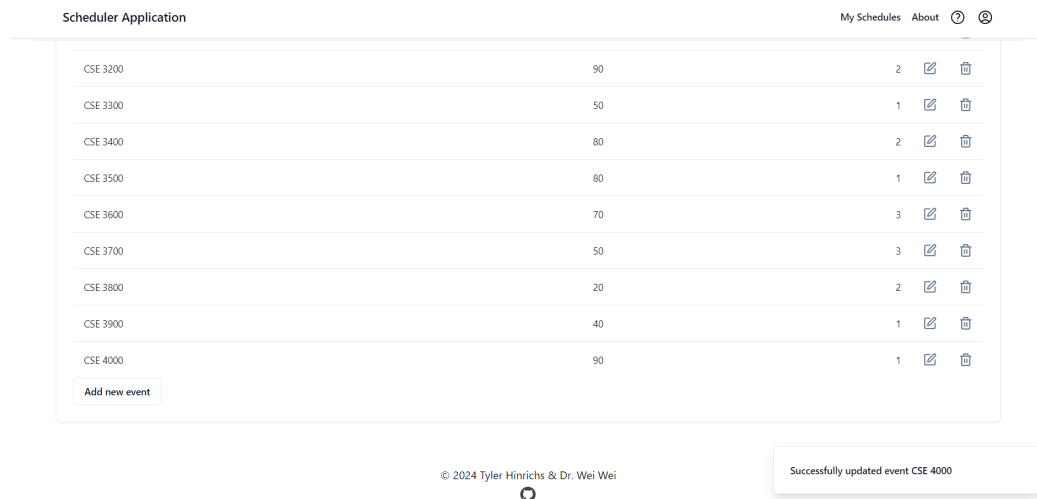


Figure 18: An example of a **Toast** appearing on the screen after the user successfully updates information about an event

ShadCN promotes responsive design, as its components are created to properly adjust to different screen sizes through Tailwind CSS. For instance, content at the `/schedules/<scheduleId>` endpoint used an adapted version of the `dashboard-01` block component provided by ShadCN to provide content to the user. When in a normal desktop view, the navigation bar in the component is laid out horizontally towards the top of the element. However, at smaller screen widths, this nav bar is swapped out for a `Sheet` element that acts as a `Dialog` that appears from the side and gives user temporary access to the menu. Both of these cases are shown below in Figure 19 and Figure 20 respectively.

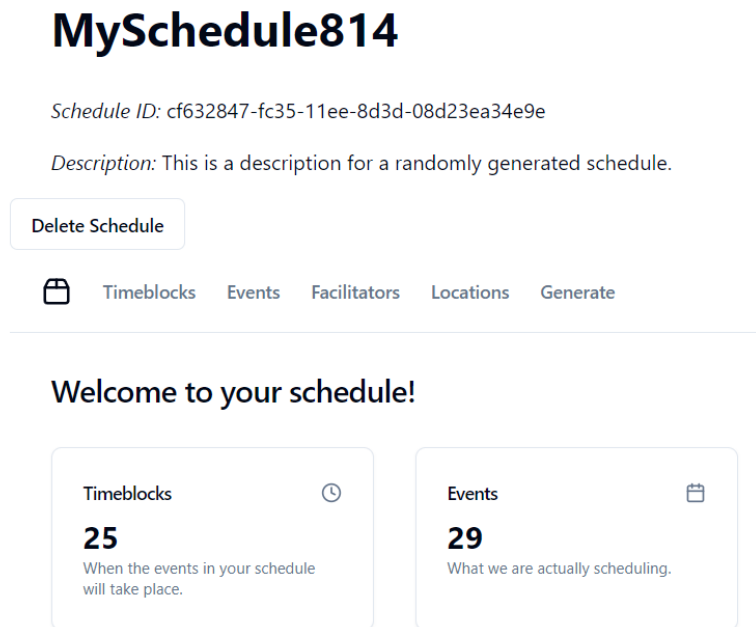


Figure 19: On a desktop format, the schedule page's navigation bar is positioned horizontally

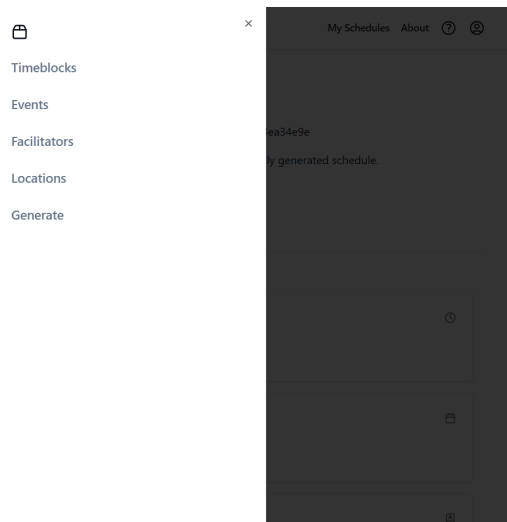


Figure 20: At smaller screen sizes, the navigation bar is placed in a Sheet, a modified Dialog

Scheduler Application

Create new timeblocks

Number of blocks per day

This is the number of blocks per day you will have in your schedule.

Number must be greater than 0

Block length

The amount of time for each block in minutes.

Minutes per block must be at least 15

Time between

The amount of time set between each timeblock in minutes. Leaving blank will default to 0.

Start time

When should your first timeblock start?

☐ Monday

Figure 21: **Form** uses React Hook Form and zod to create high-quality forms with data validation.

The **Form** component from ShadCN was also used in the application. This takes advantage of React Hook Form and zod; **Form** acts as a wrapper around the functionality of React Hook Form, while zod is used for validating data in the form. **Form** was used for generating timeblocks and for creating **ScheduleGenerations**. An example of a **Form** with data validation from zod is shown above in Figure 21.

5.3 Application Flow and Design

Next, we will investigate the user experience through a general flow that a user is intended to take when using the application.

5.3.1 Using the Application

A high-quality user experience requires close attention being paid to a user's process of using the application. The intended experience will be reviewed here, starting with a user visiting the home page of the site. There is no authentication currently implemented, but the application simulates a user being logged in. This is completed through hardcoding a `userID` in an environment variable, and then fetching its value within the Flask application and using it for subsequent database calls. When the user opens the application, they are met with content on the main page, including a help component and a schedules component that shows the schedules in their account in a table. This UI is shown in Figure 12 above.

The navigation bar at the top of the screen persists across all pages of the application. From there, clicking on “Scheduler Application” in the top links back to the base URL, whereas clicking on “My Schedules” or “About” links to `/schedules` and `/about` respectively. The question mark icon links to `/help`, and the profile icon links to `/profile`. All of these endpoints have pages with functionality that aptly corresponds to their names.

Scheduler Application

My SchedulesAbout?👤

My Profile

Update profile information.

Username

username123

First Name

Tyler

Last Name

Hinrichs

Email

example.email@uconn.edu

Update Profile Information

© 2024 Tyler Hinrichs & Dr. Wei Wei

🔄

Figure 22: UI at the `/profiles` endpoint

51

The main functionality that a user will utilize is at the `schedules/<scheduleinstanceid>` endpoint. On this screen, there is a component called `ScheduleDashboard` that hosts components that allow users to view and modify information about their schedules. The `ScheduleDashboard` component has its own navigation bar that has six different options, each of which swap out the nested component that holds main content in `ScheduleDashboard`. `ScheduleDashboard` also displays the name of the schedule, the its ID, its description, and has a button for deleting the schedule.

`ScheduleDashboard` first opens when the `/schedules/<scheduleId>` endpoint is hit. The UI welcomes the user to the schedule, with several cards showing schedule metadata, such as the number of timeblocks or events the user has added so far to the schedule. This acts as visual for a user to quickly understand how much data they have added so far, and hence the completeness of their schedule.

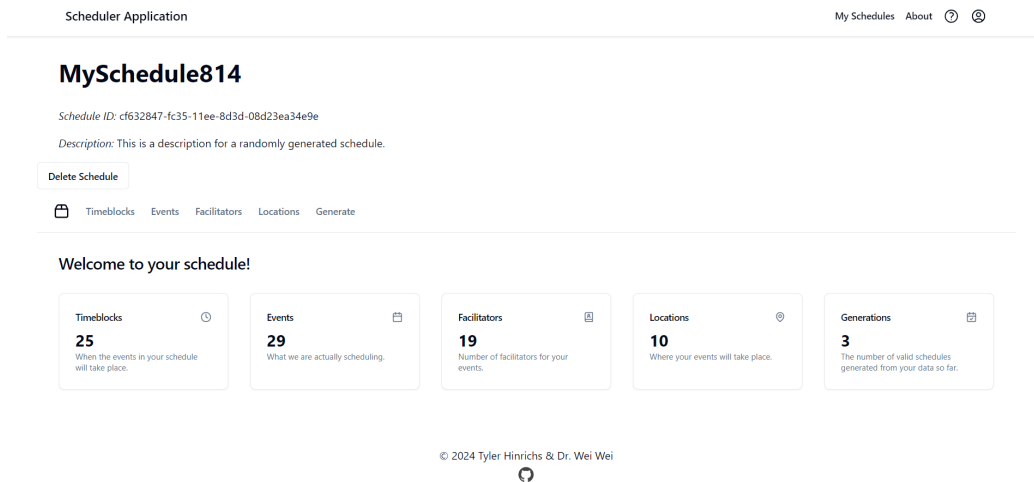


Figure 23: UI at the `/schedules/<scheduleId>` endpoint

The ordering of the nav bar in `ScheduleDashboard` was intentional; users should first specify timeblocks, as that entity forms the basis for when events can be held and when facilitators are available. Events create the entire basis for what is being scheduled, and in our schema, each is designated a user-defined weekly allotment of timeblocks. Therefore, we need timeblocks to be created prior to created events. After creating timeblocks, we create events, and following that, the order of creating locations and facilitators is relatively inconsequential since these two entities do not rely on each other in our schema.

Therefore, the timeblocks menu option was shown next in the nav bar of `ScheduleDashboard`. When clicking this option, we are taken to `/schedules/<scheduleId>/timeblocks`. On this page, there is a table showing what timeblocks currently exist for the given schedule. Below this, there is a form that gives the user options to generate timeblocks; they are given options such as number of blocks per day, block length, and options to

select which days to include. The form standardizes timeblocks to have consistent timing and spacing. This is crucial in the schema as it allows users to select how many timeblocks an event should take up, a concept that requires standardization of a timeblock in the given schedule instance. The user can later delete timeblocks, but adding timeblocks can only be completed through this form.

Scheduler Application

My Schedules About ? @

IMPORTANT

Timeblocks are a foundational part of your schedule. Creating these before working with Facilitators will make your setup process easier.

Timeblocks

Scheduled timeblocks for your schedule.

Day

Start Time

End Time

Monday

09:00

10:00

Monday

10:00

11:00

Monday

11:00

12:00

Monday

12:00

13:00

Monday

13:00

14:00

Tuesday

09:00

10:00

Tuesday

10:00

11:00

Figure 24: The timeblocks table at the `/schedules/<scheduleId>/timeblocks` endpoint

Scheduler Application

My Schedules About ? @

Create new timeblocks

Number of blocks per day

Enter number of blocks

This is the number of blocks per day you will have in your schedule.

Block length

Enter minutes

The amount of time for each block in minutes.

Time between

0 minutes (default)

The amount of time set between each timeblock in minutes. Leaving blank will default to 0.

Start time

--:-- -- @

When should your first timeblock start?

☐ Monday
 ☐ Tuesday
 ☐ Wednesday
 ☐ Thursday
 ☐ Friday
 ☐ Saturday
 ☐ Sunday

Select the days which timeblocks can occur.

Submit

Figure 25: The timeblock generation form at the `/schedules/<scheduleId>/timeblocks` endpoint

Next, we have components for events, facilitators, and location, all of which function similarly as nested UI elements within `ScheduleDashboard`. Each displays the data in a structured format, and gives the user the ability to edit, add, or delete data with dialogs. Their endpoints are:

- `/schedules/<scheduleId>/events`
- `/schedules/<scheduleId>/facilitators`
- `/schedules/<scheduleId>/locations`

respectively. The edit dialogs within each of these components have data prepopulated based on the current values of the entities. Furthermore, each delete dialog asks the user to confirm they want to delete the entity prior to deletion. Toasts will show indicating success or failure when an action is made, such as submitting data to create a new entity. Within the facilitators UI component, we had to account for facilitator eligibilities and facilitator availabilities. Within the dialogs allowing users to edit or add new facilitators, there were dropdown multiselect fields that allowed users to check or uncheck values for events and timeblocks that the facilitator was eligible for or available for respectively. Examples of each of these components are shown below.

Scheduler Application

My Schedules About ?

Timeblocks

Events

Facilitators

Locations

Generate

IMPORTANT

Events are a foundational part of your schedule. Creating these before working with Facilitators will make your setup process easier.

Events

All events for your schedule.

Event Name	Max Attendance	Number of Blocks		
CSE 1200	30	3		
CSE 1300	30	1		
CSE 1400	60	3		
CSE 1500	80	2		
CSE 1600	80	1		
CSE 1700	100	2		

Figure 26: UI at the `/schedules/<scheduleId>/events` endpoint

Facilitators

Edward Staton

Title

Professor

Bio

Good prof

Eligible Events

CSE 1200, CSE 1300, CSE 1400...

Time Avails

1, 2, 7, 17, 25

Beth Holbert

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 1400, CSE 2500, CSE 2600...

Time Avails

3, 13

Shirley Spicer

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 1700, CSE 2900

Time Avails

3, 22, 25

Steven Monzo

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 3300, CSE 3500, CSE 3800

Time Avails

4, 19, 23

Judi Harvey

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 1300, CSE 3200, CSE 3700

Time Avails

11, 22

Joan Music

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 3100

Time Avails

10, 14

Scott Owens

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 2900, CSE 3500

Time Avails

3

Dorothy Gonzales

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

No eligible events

Time Avails

1, 7

Wanda Baez

Title

Professor

Bio

This Professor is very skillful at...

Eligible Events

CSE 1200, CSE 2400, CSE 4000

Time Avails

16, 19

Edna Collard

Title

Professor

Bio

This Professor is very skillful at...

Jessie Collman

Title

Professor

Bio

This Professor is very skillful at...

Emily Spain

Title

Professor

Bio

This Professor is very skillful at...

Figure 27: UI at the `/schedules/<scheduleId>/facilitators` endpoint

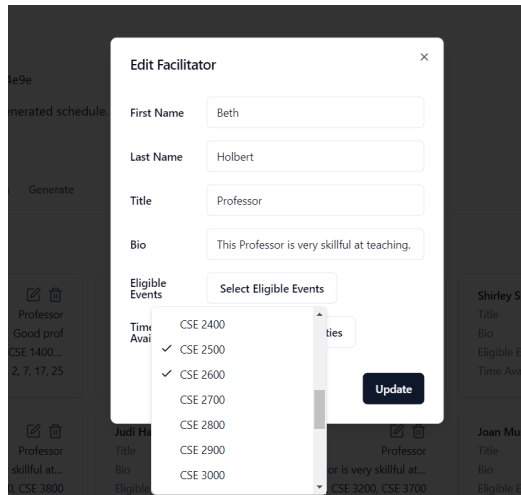


Figure 28: Dialog UI at the `/schedules/<scheduleId>/facilitators` endpoint that accounts for `facilitatoreligibleevents` and `facilitatortimeavails`





















Locations			
All locations for your schedule.			
Location Name (building)	Location Identifier (room)	Max Capacity	
HERB	1982	90	 
MCHU	286	80	 
ITE	124	100	 
HERB	281	50	 
MCHU	313	40	 
ITE	287	40	 
HERB	193	80	 
MCHU	142	40	 
ITE	204	110	 
HFRR	384	60	 

Figure 29: UI at the `/schedules/<scheduleId>/locations` endpoint

Finally, the `/schedules/<scheduleId>/generations` endpoint has a form for entering a new name and description for the current schedulegeneration, and a table that displays all schedulegenerations created so far. Selecting a schedulegeneration from the table displays each event and its details after being validly scheduled.

5.3.2 Example Schedules

Two example schedules were created for the application. Having concrete examples allows for first-time users to understand how a real-world scheduling problem could be solved with the application.

The first example schedule is the TA problem defined previously where teaching assistants of a university computer science course are being scheduled to instruct weekly lab sections. The second example is an example academic conference example, where we are trying to schedule specific events over a week timespan. This example has fewer timeblocks, but more events, facilitators, and locations.

Both of these examples attempt to illustrate real-world scenarios where administrative scheduling is crucial to fit many different constraints. Users are given these preloaded schedules when first using the application so they can interact with an example prior to using the application on their own. When they are done with the examples, they can delete them, and use the application as they desire.

The TA example is meant to directly mimic the real-world problem described previously where the CSE department at UConn was intending to schedule teaching assistants for sections of CSE 1010, the introductory com-

puter science course at the university. In this example, there are 23 teaching assistants, 40 lab sections, and timeblocks in 1 hour and 50 minute increments on Tuesday, Wednesday, Thursday, and Friday. If a user had been making this example from scratch, they would first enter the timeblocks in the aforementioned increments, and then create each event (in the spreadsheet shown in Figure 2, they are called names like 001L, 002L, etc.). All TAs should be eligible for all events, since all events are for the lab of the same course. Each TA’s availabilities should be set to the sections they specified they were available for in the spreadsheet shown in Figure 2. All data should be entered in each corresponding page as specified in the user workflow. The schedule titled “[Example] Teaching Assistant Scheduling” shows what would show up in the application if the user had already entered data for this example themselves. Then, they can navigate to the generate tab and run the algorithm to find a valid instance of the schedule.

6 Conclusion and Future Work

The goal of this research was to create a web-based scheduling application that approached modeling a multitude of scheduling problems generically. This task was completed through several different key aspects. The foundation for the scheduling algorithm was based on previous techniques where the genetic algorithm was used for university course scheduling [1]. After working with the genetic algorithm in several simple examples, the idea of a generic scheduling scenario was introduced. This generic scenario was represented in a logical schema that was iterated upon and later turned into a relational schema in MySQL database. After the schema and MySQL database were created, demo data was created and added to the database. The genetic algorithm created for the TA solution was adapted to support the inclusion of new entities in the schema. Following this, an API was made with Python and Flask to give access to objects in the database over HTTP using REST APIs. Within Flask, Flask-SQLAlchemy was used to map entities in the database to objects that could be usable in the Python-based algorithm. Finally, an intuitive UI was designed using the component library React alongside modern components provided by ShadCN UI that resulted in a functional, robust application.

There are several areas to expand upon in future work for this application. First, for the application to be used by multiple users, authentication should be supported. Users would be able to sign on and have access to scheduling information unique to their account.

Furthermore, FastAPI could be explored as an API option; it is a frame-

work similar to Flask, but has purported efficiency advantages. This option could be explored if efficiency became a concern in this application.

Furthermore, form logic should be standardized. Currently, there are two paradigms in the application; most dialogs use input elements from ShadCN and do not use zod for form validation. Instead, requests are made to the backend, and if the data is invalid, an error message is returned to the client and displayed. However, for the forms at

- `schedule/<scheduleId>/timeblocks`
- `schedule/<scheduleId>/generate`

both use `Form` provided by ShadCN UI as well as zod for client-side data validation. This prevents making unnecessary requests when we already know that they would fail from the client perspective.

The current schema intentionally limits the flexibility of timeblocks in order to standardize facilitator availabilities and allow users to specify the number of timeblocks each event should take. This could potentially be expanded to be more flexible in the future.

One UI change to consider is representing both timeblocks and scheduledevents visually with regards to chronological order and location. Scheduling is most easily understood when a user can picture representations of timings and their relation to one another. In future iterations, a user could view the timeblocks they are adding or removing on a weekly calendar. Showing scheduledevents visually could help users see when each event could potentially occur in relation to one another; because there could be multiple

locations, different schedules for different locations could be shown.

The process of manually entering or retrieving data from the application is currently inefficient, so it would improve user experience to give the option to upload and download information via CSV files or another format.

Ultimately, the achievements made through this project do not need to end here, as there are still many areas to improve upon. Future iterations should look to standardize errors and make the API more robust. User testing and iterating on user feedback are crucial next steps to ensure that the application meets real-world standards. Future development should prioritize adaptability to meet the evolving needs of organizations with unique administrative scheduling tasks.

References

- [1] Athina, J., & Wei, W. (2023). Improving Web-Based Scheduling Systems: A Machine Learning Approach.
- [2] Bernstein, P. A., Copeland, G., Maier, D., Ambler, S., Fowler, M., Hohpe, G., Atzeni, P., Yoder, J. W., Buff, H. W., Ambler, S. W., Brown, K., Keller, W., Neward, T., Atkinson, M. P., Atkinson, M., Gamma, E., Berler, M., Karwin, B., Chen, T.-H., & Lämmel, R. (2016, September 28). *Twenty Years of object-relational mapping: A survey on patterns, solutions, and their implications on application design*. Information and Software Technology. <https://www.sciencedirect.com/science/article/abs/pii/S0950584916301859>
- [3] Ellingwood, J. (2023). Comparing database types: how database types evolved to meet different needs. Prisma's Data Guide. <https://www.prisma.io/dataguide/intro/comparing-database-types>
- [4] Gómez, F. (2023, June 29). *Genetic algorithms for feature selection in machine learning*. Neural Designer. https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection/
- [5] Katoch, S., Chauhan, S. S., & Kumar, V. (2020, October 31). A review on genetic algorithm: past, present, and future. Springer-Link. <https://link.springer.com/article/10.1007/s11042-020-10139-6> Introduction§ion=ControlPID (accessed Dec. 17, 2022).
- [6] Kumar, K., & Azad, S. K. (2017). Database normalization design pattern. <https://ieeexplore.ieee.org/Xplore/home.jsp>

- [7] MathWorks. (2023). Genetic Algorithm. MathWorks Help Center. <https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html>
- [8] Meta Open Source. (2024). React. <https://react.dev/>
- [9] OCI. (2023). What is MySQL?. Oracle. <https://www.oracle.com/mysql/what-is-mysql/>
- [10] Pallets. (2023). *Flask*. Welcome to Flask - Flask Documentation (3.0.x). <https://flask.palletsprojects.com/en/3.0.x/>
- [11] Pallets. (2024). *Flask SQLAlchemy*. Flask. <https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/>
- [12] Rawat, P., & Mahajan, A. N. (2020, November). ReactJS: A Modern Web Development Framework. <https://ijisrt.com/assets/upload/files/IJISRT20NOV485.pdf>
- [13] Remix Software, Inc. (2024). *React Router*. Home v6.22.0. <https://reactrouter.com/en/main>
- [14] Rodríguez, C. *et al.* (2016). REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds) Web Engineering. ICWE 2016. Lecture Notes in Computer Science(), vol 9671. Springer, Cham. https://doi.org/10.1007/978-3-319-38791-8_2
- [15] shadcn. (2024). *ShadCN Docs*. shadcn/ui. <https://ui.shadcn.com/docs>

- [16] Singh, G. (2023, April 5). *What are State and uses of state in react*. Webkul Blog. <https://webkul.com/blog/react-state/>
- [17] SQLAlchemy. (2024). *Sqalchemy*. SQLAlchemy. <https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/>
- [18] Teorey, T., & Jagadish, H. V. (2011). *Impedance Mismatch*. Impedance Mismatch - an overview — ScienceDirect Topics. <https://www.sciencedirect.com/topics/computer-science/impedance-mismatch>
- [19] The Builder. (2020, September 10). Genetic Algorithm In Python Super Basic Example. YouTube. <https://www.youtube.com/watch?v=4XZoVQOt-0I>
- [20] Zolotarev, J. (2022, December 19). *What Is React?*. Built In. <https://builtin.com/software-engineering-perspectives/react>