CSDS 341, Project Final Report
Nicholas Kernan, Noel Mathew, Victor Wang, David Ye
May 5th, 2021

Nonprofit Information Management Service

1. Introduction: Overview

The purpose of this system is to maintain information that would be useful for nonprofit organizations specifically regarding managing and record-keeping of staff, finances, and donors.

Nonprofit organizations (NPOs) are established not with the intent of generating revenue and profits but for increasing the welfare of society. Examples of well-known NPOs are the Bill and Melinda Gates Foundation and American Red Cross. In the United States, the Internal Revenue Service gives NPOs a 501(c)(3) status which exempts them from paying taxes. In order to maintain their tax code, NPOs must have well-documented and transparent records of their finances. Other problems that nonprofits face include fraud prevention and finding new donors. While nonprofits do have some similarities to for-profit businesses, a standard for-profit database can not be used for nonprofit organizations because of their unique internal structure primarily highlighted by their composition of paid employees and unpaid volunteers at every level of management in the organization.

Through our nonprofit information management database, it will be possible to effectively store a large volume of information regarding multiple nonprofit organizations and make it possible to easily query allowed information to relevant users. The database will be used by both the NPO as well as the public. NPOs can use the service to manage their employee payrolls and contact information, keep track of information relevant to financial reports and IRS forms (such as donor information). Additionally, current and potential donors can use the service to see how funds are managed to evaluate the effectiveness of the NPO. Additionally, as NPOs often both receive and distribute charitable donations, the purpose of our project is to design a database structure to maintain and catalogue these donations, both lump-sum and recurring.

2. Database Requirement Specifications

2.a Data Description

Objects:

Nonprofit
An instance of this object will represent a single nonprofit organization. It will store information about the nonprofit organization, such as a name, address, email and a unique id.

Donor
This represents people who will be or have given money to a nonprofit organization. Donor will store their name, address, gender, email and unique id.

Expense
This represents something a nonprofit organization would spend money on. The dollar amount is kept as well as what date the expense was incurred.

Employee
This represents the workers at a nonprofit. Their name, gender, phone number and a unique id will be kept. Also, each employee will be linked to a particular department.

Department
This represents a department within a nonprofit organization such as budgeting or fundraising. Department will store the name of the department, the director of the department, and the budgets of the department.

Volunteer
This represents an unpaid volunteer who worked for the nonprofit organization. Volunteer will store a unique id for the volunteer, the name of the volunteer, the gender of the volunteer, and the email, address and phone number of the volunteer.

Relationships:

Donation
This relates donors to nonprofits, through donations they have made. Donations will store the id of the donor and nonprofit, the dollar amount of the donation and what date it occurred on. It will also have a “recurrence” - since donations may be one-time, monthly or yearly.

Pledge

This is very similar to donations. The difference is that pledges will keep track of information about donations that are scheduled to occur in the future, whereas donations maintain the donations that have already occurred in the past.

Npo_Dept
This relates departments to the nonprofit organizations they are a part of.

Npo_Exp
Each expense belongs to a nonprofit organization, so this simply relates the two.

Employs
This relates employees to the department that is employing them, and stores the salary of the employee.

VolunteerHours
This relates volunteers to the nonprofit organizations they have done volunteering for. It will store the date of the volunteering and number of hours worked.

Budget
This represents the amount of money allocated to a particular department of a nonprofit organization in a particular year.

2.b Queries

A nonprofit organization may wish to keep track of its finances by seeing what donations and expenses it has made recently. It can do so by querying those relations using its nonprofit id. It may also wish to compare the donations that have occurred recently with the pledges that are scheduled to occur soon, to estimate whether donations to the organization are picking up or declining. It can thus query donations and pledges with its nonprofit id and the relevant dates. These types of queries would likely be pretty frequent.

Donors may wish to keep track of their finances by seeing what upcoming donations they have agreed to and whether it is more or less than they intended. They could query pledges using their donor id. If they are donating to a particular organization, they may wish to support organizations with a similar cause. This could be done with a complex query that would find organizations that are commonly donated to by the same donors who donate to the organization in question (as presumably, the other commonly donated to organizations would have a similar cause).

Finally, government entities and regulatory agencies may use the database to monitor behavior of nonprofit organizations. For example, they may query employees to see if

there is gender equality in payment. They may also query donations, expenses and budgets to find the overall surplus/deficit for a nonprofit, as it may affect their tax status. These types of queries would likely occur about once a year for each nonprofit.

2.c. Integrity Constraints

1. The Nonprofit and donor relations will have an id attribute, that will be unique for all instances of that relation. For example, no two nonprofits can have the same “nonprofit id”.

2. Attributes that are clearly references to other relations need to actually exist in those relations. For example, “vol_id” in VolunteerHours references Volunteer, so all vol_id’s that appear in VolunteerHours need to exist in Volunteer.

3. There can only be one instance of currentDate.

4. All dates in “donations” will be before the current date, and all dates in “pledges” will be on or after the current date.

5. When the current date is updated, all entries in “pledges” that are now at or before the current date will be copied to “donations”, and deleted from “pledges”.

6. When a pledge is turned into a donation or a new donation is created, if its recurrence attribute is “monthly”, a new pledge will be created with the same entries except the date will be set for a month afterwards (with this rule, a new pledge will again be created when that one becomes a donation, and so on). The same rule applies if the recurrence is “yearly”, only the pledge will be created for a year afterwards.

7. A monthly donation cannot be inserted where it’s date is more than a month before the current date. Similarly, a yearly donation cannot be inserted from more than a year before the current date.

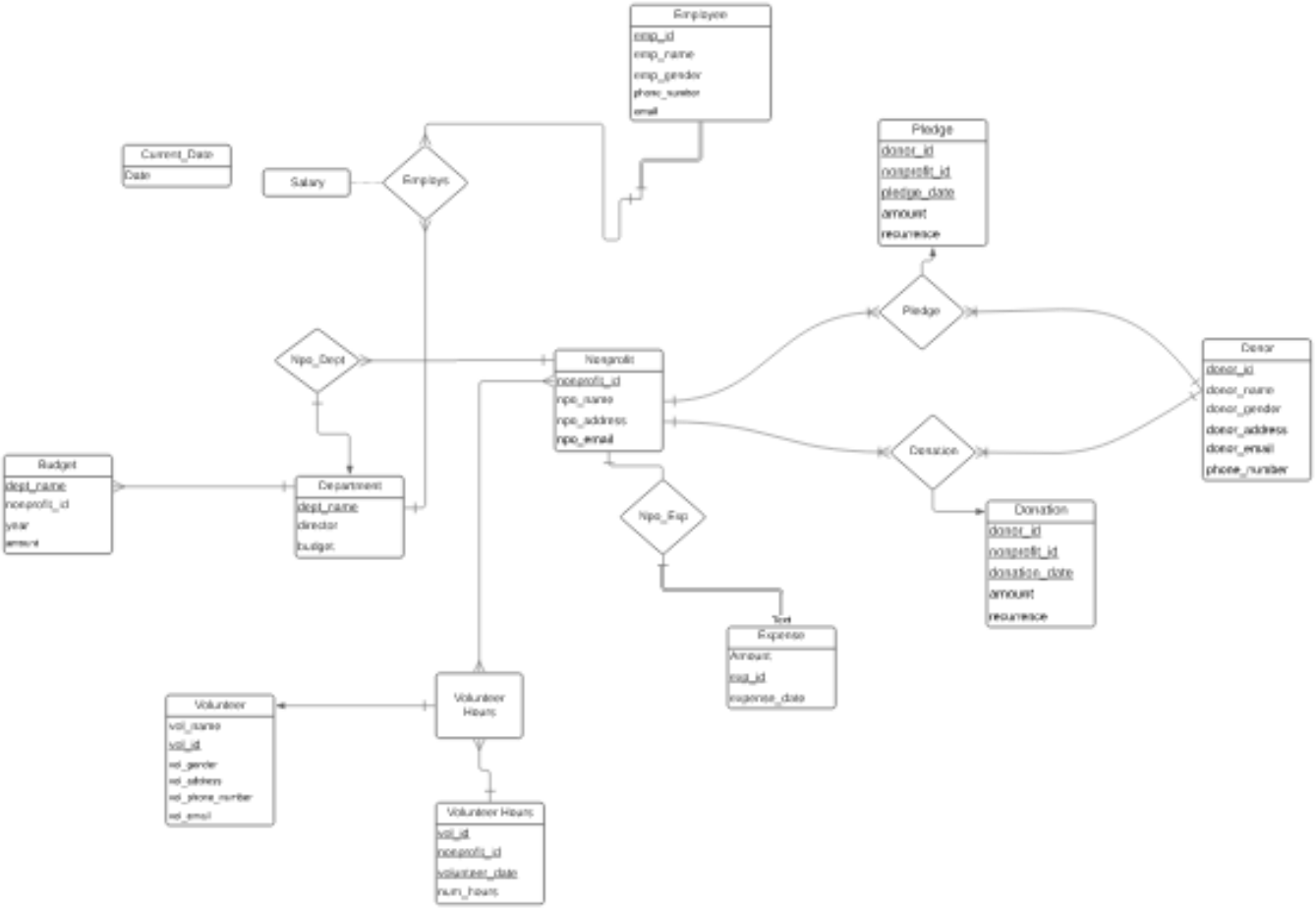
8. CurrentDate cannot change by more than a month at a time.

9. A single donation cannot have a dollar amount greater than 1 million dollars.

10. The combined salaries of employees in a department cannot be greater than the budget of that department.

11. No entry for “volunteer hours” can be greater than 24 hours.

3. ER Data Model Design



Entities:

All of the below entities are strong entities except Department, since dept_name needs to be associated with nonprofit_id from npo_dept to uniquely identify a Department. Primary keys are underlined.

Nonprofit(nonprofit_id, npo_name, npo_address, npo_email)

Donor (donor_id, donor_name, donor_gender, donor_address,
donor_phone_number, donor_email)

Employee(emp_id, emp_name, emp_gender, emp_address, phone_number,
emp_email, salary)

Expense(exp_id, expense_date, amount)

Department(dept_name, director, budget)

Volunteer(vol_id, vol_name, vol_gender, vol_address, vol_phone_number, vol_email)

Current_Date(date)

Relationships:

Pledge(donor_id, nonprofit_id, pledge_date, amount, recurrence)
-Many to many between donors and nonprofits

Donation(donor_id, nonprofit_id, donation_date, amount, recurrence)
-Many to many between donors and nonprofits

npo_exp(nonprofit_id, exp_id)
-Relates one nonprofit to many expenses

employs(nonprofit_id, dept_name, emp_id, salary)
-Relates one department to many employees

npo_dept(nonprofit_id, dept_id)
-Relates one nonprofit to many departments

VolunteerHours(vol_id, nonprofit_id, volunteer_date, num_hours)
-Many to many between volunteers and nonprofits

Domains:
All ids, names, addresses and emails will be varchars. Phone numbers, num_hours and dollar amounts (amount, salary, budget) will be numeric. We have two enums: gender (‘male’, ‘female’) and recurrence (‘one-time’, ‘monthly’, ‘yearly’). Enums are internally

stored as varchars, with a constraint limiting them to a certain number of values. This is because SQL Server does not natively have an enum type.
Finally, the date attributes are simply of the date type.

Properties:
Dates are composite attributes, consisting of a year, month and day.
Budget in department is multivalued. A department may have a different budget for each year.
All other attributes are simple and single-valued.

4. Transforming the ER Model into the Relational Model

When converting from the E-R model to the relational model, the donation and pledge relationships need to be represented in their own relations as they are many to many (and therefore cannot simply be added to nonprofit or donor). For both of them, primary keys are formed with (donor_id, nonprofit_id, date). This is because while the same donor may donate to the same nonprofit many times, it is assumed that they will not do so within the same date. Therefore, this collection of attributes will uniquely represent any donation/pledge.

Similarly, because VolunteerHours is many-to-many, this relationship needs to become a relation. To associate volunteers with nonprofits, VolunteerHours has vol_id and nonprofit_id as foreign keys. These attributes along with volunteer_date will form a primary key. If a volunteer works with the same nonprofit multiple times during the same day, they can simply add their hours together in “num_hours” so only one entry is needed.

The npo_exp and npo_emp do not need to be turned into relations, because each employee or expense will be associated with exactly one nonprofit organization. Therefore, when Expense and Employee are turned into relations, nonprofit_id can be added as a foreign key to both of them.

Since budget is multivalued, a new relation is created for budget. It will store a dept_id, year and dollar amount. Thus, by querying budget with a dept_id one can obtain its budget for every year.

Composite attributes usually have to be split up when converting to the relational model, but the date type in SQL allows for encapsulation/comparison of dates without needing to split day, month and year into their own attributes.

Finally, Board of Directors does not actually need to become a relation, because all the directors in a nonprofit organization can be found by querying Department.

Relations:

Nonprofit(nonprofit_id, npo_name, npo_address, npo_email)

Donor (donor_id, donor_name, donor_gender, donor_address,
donor_phone_number donor_email)

Pledge(donor_id, nonprofit_id, pledge_date, amount, recurrence)

Donation(donor_id, nonprofit_id, donation_date, amount, recurrence)

Employee(emp_id, dept_name, nonprofit_id, emp_name, emp_gender, emp_address, phone_number, emp_email, salary)

Expense(exp_id, nonprofit_id, expense_date, amount)

Department(dept_name, director, nonprofit_id)

Budget(dept_name, nonprofit_id, year, amount)

Volunteer(vol_id, dept_name, nonprofit_id, vol_name, vol_gender, vol_address,
vol_phone_number, vol_email)
VolunteerHours(vol_id, nonprofit_id, volunteer_date, num_hours)

Current_Date(date)

5. Creating Our Database

Our choice of DBMS was Microsoft SQL Server. While not as popular as MySQL, we chose this DBMS because of its native compatibility with VSCode, our IDE of choice when working with both SQL and Java. Additionally, using SQL Server allowed us to use Microsoft Azure, which had the bonus effect of allowing us to maintain our entire database remotely, so that all members of our team could work on the database simultaneously.

A minor issue with using SQL Server was that some of the syntax for creating triggers was different than in the book. For example, “Create trigger recurringDonations After insert on Donation” was not accepted by SQL Server, and needed to be changed to “Create trigger recurringDonations on Donation after insert”.

```
CREATE TABLE Nonprofit
(
        nonprofit_id VARCHAR(10) NOT NULL PRIMARY KEY,
        npo_name VARCHAR(50) NOT NULL,
        npo_address VARCHAR(50) NOT NULL,
```

```
	npo_email VARCHAR(50) NOT NULL
);

CREATE TABLE Donor
(
	donor_id VARCHAR(10) NOT NULL PRIMARY KEY,
	donor_name VARCHAR(50) NOT NULL,
	donor_gender VARCHAR(6) NOT NULL CHECK (donor_gender IN ('male',
'female')),
	donor_address VARCHAR(50) NOT NULL,
	donor_phone_number NUMERIC(11, 0) NOT NULL,
	donor_email VARCHAR(50) NOT NULL
);

CREATE TABLE Donation
(
	donor_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES Donor,
	nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
	donation_date DATE NOT NULL,
	amount NUMERIC(9, 2) NOT NULL CHECK(amount <= 1000000),
	recurrence VARCHAR(10) NOT NULL CHECK (recurrence IN ('one-time',
'monthly', 'yearly')),
	CONSTRAINT PK_donation PRIMARY KEY (donor_id, nonprofit_id,
donation_date)
);


CREATE TABLE Pledge
(
	donor_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES Donor,
	nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
	pledge_date DATE NOT NULL,
	amount NUMERIC(9, 2) NOT NULL CHECK(amount <= 1000000),
	recurrence VARCHAR(10) NOT NULL CHECK (recurrence IN ('one-time',
'monthly', 'yearly')),
	CONSTRAINT PK_pledge PRIMARY KEY (donor_id, nonprofit_id, pledge_date)
);

CREATE TABLE Department
(
	dept_name VARCHAR(50) NOT NULL,
	director_name VARCHAR(50),
	nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
```

```
	CONSTRAINT PK_dept PRIMARY KEY (dept_name, nonprofit_id)
);

CREATE TABLE Budget
(
	dept_name VARCHAR(50) NOT NULL,
	nonprofit_id VARCHAR(10) NOT NULL,
	FOREIGN KEY(dept_name, nonprofit_id) REFERENCES Department,
	year NUMERIC(4, 0) NOT NULL,
	amount NUMERIC(9, 2) NOT NULL
);

CREATE TABLE Employee
(
	emp_id VARCHAR(10) PRIMARY KEY,
	dept_name VARCHAR(50) NOT NULL,
	nonprofit_id VARCHAR(10) NOT NULL REFERENCES Nonprofit,
	emp_name VARCHAR(50) NOT NULL,
	emp_gender VARCHAR(6) NOT NULL CHECK (emp_gender IN ('male',
'female')),
	emp_address VARCHAR(50) NOT NULL,
	emp_phone_number NUMERIC(11, 0) NOT NULL,
	emp_email VARCHAR(50) NOT NULL,
	salary NUMERIC(9, 2) NOT NULL,
	CONSTRAINT FK_dept_emp FOREIGN KEY (dept_name, nonprofit_id)
REFERENCES Department (dept_name, nonprofit_id)
);

CREATE TABLE Expense
(
	amount NUMERIC(9, 2) NOT NULL CHECK(amount <= 1000000),
	exp_id VARCHAR(10) PRIMARY KEY,
	nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
	expense_date DATE NOT NULL
);

CREATE TABLE Volunteer
(
	vol_id VARCHAR(10) NOT NULL PRIMARY KEY,
	dept_name VARCHAR(50) NOT NULL,
	nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
	vol_name VARCHAR(50) NOT NULL,
	vol_gender VARCHAR(6) NOT NULL CHECK (vol_gender IN ('male', 'female')),
	vol_address VARCHAR(50) NOT NULL,
```

```
        vol_phone_number NUMERIC(11, 0) NOT NULL,
        vol_email VARCHAR(50) NOT NULL,
        CONSTRAINT FK_dept_vol FOREIGN KEY (dept_name, nonprofit_id)
REFERENCES Department (dept_name, nonprofit_id)
);

CREATE TABLE VolunteerHours
(
        vol_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES Volunteer,
        nonprofit_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES
Nonprofit,
        volunteer_date DATE NOT NULL,
        num_hours NUMERIC(2, 0) NOT NULL CHECK(num_hours < 24)
        CONSTRAINT PK_volunteerhours PRIMARY KEY (vol_id, nonprofit_id,
volunteer_date)
);

CREATE TABLE CurrentDate
(
        date DATE NOT NULL,
);
```

6. SQL Queries:

Query 1: Find the names of Nonprofit organizations that the donor "John Adams" has made pledges to.

This is a very simple query that would be very useful for a donor to find out what organizations they have promised to donate to in the near future, and help them manage their budget.

SQL:

```
SELECT Nonprofit.npo_name
FROM Nonprofit, Pledge, Donor
WHERE Nonprofit.nonprofit_id = Pledge.nonprofit_id
AND Pledge.donor_id = Donor.donor_id
AND Donor.donor_name = 'John Adams'
```

Relational Algebra:

π_{npo_name} (Nonprofit $\bowtie_{nonprofit\_id}$ (Pledge $\bowtie_{donor\_id}$ ($\sigma_{donor\_name='John\ Adams'}$(Donor))))

TRC:
{t | (∃ n) (∃ p) (∃ d) (Nonprofit(n)∧Pledge(p) ∧ Donor(d) ∧ t[npo_name] = n[npo_name] ∧ n[nonprofit_id] = p[nonprofit_id] ∧ p[donor_id] = d[donor_id] ∧ d[donor_name] = 'John Adams') }

Query 2: Find the Board of Directors (the names and departments of all department directors) for the "American Red Cross".

This is another simple query. It may be useful for an employee at the American Red Cross, who may need to make some type of proposal to their Board of Directors.

SQL:
SELECT dept_name, director_name

FROM Department D, Nonprofit N

WHERE N.npo_name = 'American Red Cross'

and D.nonprofit_id = N.nonprofit_id

Relational Algebra:
$\pi_{\text{dept\_name, director\_name}}$ (Department $\bowtie_{\text{nonprofit\_id}}$ ($\sigma_{\text{npo\_name='American Red Cross'}}$(Nonprofit)))

TRC:
{t |(∃n)(∃d) (Nonprofit(n)∧Department(d) ∧ t[dept_name] = d[dept_name] ∧ t[director_name] = d[director_name] ∧ n[npo_name] = 'American Red Cross' ∧ d[nonprofit_id] = n[nonprofit_id]) }

Query 3: Find the email addresses of donors who have made pledges or donations for the Nonprofit 'American Red Cross', but have not made pledges or donations to the Nonprofit 'Doctors without Borders'

This may be a very useful query for "Doctors_without_Borders" to try to find and contact new donors: donors who are already donating to the "Red_Cross" are probably more willing to donate to "Doctors_without_Borders" as they serve similar causes.

SQL:
SELECT D.donor_email

FROM Donor D, Pledge, Donation, Nonprofit
WHERE
((D.donor_id = Pledge.donor_id AND Pledge.nonprofit_id = Nonprofit.nonprofit_id)
OR
(D.donor_id = Donation.donor_id AND Donation.nonprofit_id = Nonprofit.nonprofit_id)
)AND Nonprofit.npo_name = 'American Red Cross'
AND **NOT EXISTS**
(**SELECT** *
FROM Pledge P, Donation Don, Nonprofit N
WHERE
((D.donor_id = P.donor_id AND P.nonprofit_id = N.nonprofit_id)
OR
(D.Donor_id = Don.donor_id AND Don.nonprofit_id = N.nonprofit_id))
AND N.npo_name = 'Doctors Without Borders')

Relational Algebra:

π_{donor_email} (

((Nonprofit $\bowtie_{nonprofit\_id}$ (Donation $\bowtie_{donor\_id}$ ($\sigma_{npo\_name='American\ Red\ Cross'}$(Nonprofit)))) $\cup$
(Nonprofit $\bowtie_{nonprofit\_id}$ (Pledge $\bowtie_{donor\_id}$ ($\sigma_{npo\_name='American\ Red\ Cross'}$(Nonprofit)))))
-
((Nonprofit $\bowtie_{nonprofit\_id}$ (Donation $\bowtie_{donor\_id}$ ($\sigma_{npo\_name='Doctors\ Without\ Borders'}$ (Nonprofit))))
$\cup$(Nonprofit $\bowtie_{nonprofit\_id}$ (Pledge $\bowtie_{donor\_id}$ ($\sigma_{npo\_name='Doctors\ Without\ Borders'}$(Nonprofit))))))

TRC:
{t | ($\exists$d) (Donor(d) $\wedge$ t[donor_email] = d[donor_email] $\wedge$ ($\forall$p)($\forall$don)($\forall$n) (Pledge(p) $\wedge$ Donation(don) $\wedge$ Nonprofit(n) $\wedge$ n[npo_name] = 'American Red Cross' $\wedge$
((d[donor_id] = don[donor_id] $\wedge$ don[nonprofit_id] = n[nonprofit_id]) $\vee$ (d[donor_id] = p[donor_id] $\wedge$ n[nonprofit_id] = p[nonprofit_id])) $\rightarrow$ $\neg$(($\exists$p1)($\exists$don1)($\exists$n1))
(n1[npo_name] = 'Doctors Without Borders' $\wedge$((d[donor_id] = don1[donor_id] $\wedge$ don1[nonprofit_id] = n1[nonprofit_id]) $\vee$ (d[donor_id] = p1[donor_id] $\wedge$ n1[nonprofit_id] = p1[nonprofit_id]))) }

}

Query 4: Find the names of Nonprofit organizations who, on average, pay their female employees 88 cents or less for every dollar that their male employees make.

This could be a very useful query for regulatory agencies or justice groups, who want to make sure that there is equality in the compensation between men and women.

```
SELECT N.npo_name
FROM Nonprofit N
WHERE (select AVG(salary)
      from employee E
where E.nonprofit_id = N.nonprofit_id and E.emp_gender = 'female')
<= .88 * (select AVG(salary)
```

Query 5: Find the names of nonprofit organizations where the amount of money pledged to the nonprofit organization is greater than the amount of money that has already been donated.

This is useful for identifying up and coming nonprofit organizations who are likely in their infancy yet are gaining a lot of attention from donors.

```
SELECT Nonprofit.npo_name
FROM Nonprofit
WHERE ( ( SELECT sum(Pledge.amount)
        FROM Pledge
        WHERE nonprofit.nonprofit_id = Pledge.nonprofit_id)
>
( SELECT sum(Donation.amount)
        FROM Donation
        WHERE nonprofit.nonprofit_id = Donation.nonprofit_id) )
```

Query 6: Find the names of nonprofit organizations that have received a greater number of donations in the previous year than they are currently expected to receive next year (pledges for next year).

This will be useful to see which nonprofit organizations are potentially in decline, as they are expecting to receive less donations next year than they have the previous year. This may be a sign that they should improve their outreach efforts and try to gain more pledges.

```
SELECT Nonprofit.npo_name
FROM Nonprofit, currentDate date
WHERE ( SELECT count(*)
    FROM Donation D
    WHERE nonprofit.nonprofit_id = D.nonprofit_id
AND abs(datediff(year, D.donation_date, date)) <= 1)
>
( SELECT count(*)
    FROM Pledge P
    WHERE nonprofit.nonprofit_id = P.nonprofit_id
AND abs(datediff(year, P.pledge_date, date)) <= 1)
```

Query 7: For each month of last year, find the names of the donors who made a donation to the "American Red Cross" with the highest dollar amount of any donations to the "American Red Cross" that month.

This could be helpful for the Red Cross to find who its biggest donors have been. And if the same donor name is appearing across multiple months, they may want to send special thank-you messages to that person.

```
SELECT month(D.donation_date), Donor.donor_name
```

```
FROM Donor, Donation D, Nonprofit, currentDate date
WHERE Donor.donor_id = D.donor_id
AND D.nonprofit_id = Nonprofit.nonprofit_id
AND Nonprofit.npo_name = 'American Red Cross'
AND abs(datediff(year, D.donation_date, date)) <= 1
AND D.amount = (SELECT MAX(D1.amount)
                FROM Donation D1
                WHERE month(D1.donation_date) = month(D.donation_date)
                    AND D1.nonprofit_id = D.nonprofit_id
                    AND year(D1.donation_date) = year(D.donation_date))
```

Query 8: Find the names of volunteers who have done more than 50 hours of volunteering within a single week.

This would be a useful query for regulatory agencies, because a volunteer working more than 50 hours/week could be a major red flag. It may be a signal that the nonprofit they are volunteering for is overworking them, or that they should be getting paid.

```
SELECT vol_name
FROM volunteer V
WHERE Exists(
   Select volunteer_date
   from volunteerHours H
   where H.vol_id = V.vol_id
   and (select sum(num_hours)
      from volunteerHours H2
      where H.vol_id = H2.vol_id
and abs(datediff(day, H2.volunteer_date, H.volunteer_date)) <= 7)
> 50)
```

Query 9: For each year, find the total surplus/deficit (donations minus budgets and expenses) for the 'American Red Cross'.

This is perhaps the most important query for nonprofit organizations to keep track of their finances. Clearly, it is a red flag if they have a large deficit for most years. It is also information they will need to be reporting to government agencies to keep their tax-exempt status.

```
SELECT all_cash_flows.year, sum(all_cash_flows.amount)
FROM (select year(e.expense_date) as year, sum(amount)*-1 as amount
        from expense e, nonprofit n
        where e.nonprofit_id = n.nonprofit_id and n.npo_name = 'American Red Cross'
        group by year(e.expense_date)
union all
        select year(d.donation_date) as year, sum(amount) as amount
        from donation d, nonprofit n
        where d.nonprofit_id = n.nonprofit_id and n.npo_name = 'American Red Cross'
        group by year(d.donation_date)
union all
        select year, sum(amount)*-1 as amount
        from budget b, nonprofit n
        where b.nonprofit_id = n.nonprofit_id and n.npo_name = 'American Red Cross'
        group by b.year
) as all_cash_flows
GROUP BY year
```

7. Integrity Constraints

1. *The Nonprofit and donor relations will have an id attribute, that will be unique for all instances of that relation. For example, no two nonprofits can have the same "nonprofit id".*

This integrity constraint is enforced by simply declaring each id attribute as a primary key in their respective relations. For example, in the nonprofit relation, the line

PRIMARY KEY(nonprofit_id) enforces this constraint.

2. *Attributes that are clearly references to other relations need to actually exist in those relations. For example, "vol_id" in VolunteerHours references Volunteer, so all vol_id's that appear in VolunteerHours need to exist in Volunteer.*

This constraint is also enforced fairly easily, by declaring foreign keys. For example, the line vol_id VARCHAR(10) NOT NULL FOREIGN KEY REFERENCES Volunteer ensures that all vol_id's appear in Volunteer.

3. *There can only be one instance of currentDate.*

   This can be enforced with a trigger on insertions to currentDate that raise an error if it causes the count of relations to be 2 or greater. We can also prevent deletions to currentDate.

```
Create Trigger noDateAdd on CurrentDate
After Insert
As
if((select count(*) from currentdate) >= 2)
throw 60000, 'Do not add to currentDate, you should update it.', 1;


Create Trigger noDateRemove on CurrentDate
After Delete
As
throw 60000, 'Do not remove currentDate, you should update it.', 1;
```

4. *All dates in "donations" will be before the current date, and all dates in "pledges" will be on or after the current date.*

These integrity constraints can be enforced with triggers. First, we can rollback added donations when they are ahead of the current date, to make sure any donations added are before the current date:

```
CREATE TRIGGER donationDateCheck on donation
    AFTER INSERT
    as
    if exists(select *
```

```
            from inserted
           where inserted.donation_date >= (select * from currentdate))
    throw 70000, 'Donations need to be before the current date.', 1;
```

We can create a similar trigger to check that new insertions to pledges are after the current date.

```
CREATE TRIGGER pledgeDateCheck on pledge
    AFTER INSERT
    as
    if exists(select *
             from inserted
            where inserted.pledge_date < (select * from currentdate))
    throw 80000, 'Pledges need to be at or after the current date.', 1;
```

5. *When the current date is updated, all entries in "pledges" that are now at or before the current date will be copied to "donations", and deleted from "pledges".*

We can make a trigger that will insert all pledges that are now before the current date into donations, and then delete them from pledges.

```
CREATE TRIGGER pledgesToDonations on currentDate
    AFTER UPDATE
    as
    BEGIN
        INSERT INTO Donation
        SELECT *
        FROM Pledge P
        WHERE P.pledge_date < (select date from currentdate)

        DELETE FROM Pledge
        WHERE pledge_date < (select date from currentdate)
    END
```

6. *When a pledge is turned into a donation or a new donation is created, if its recurrence attribute is “monthly”, a new pledge will be created with the same entries except the date will be set for a month afterwards (with this rule, a new pledge will again be created when that one becomes a donation, and so on). The same rule applies if the recurrence is “yearly”, only the pledge will be created for a year afterwards.*

This is made with a trigger on insertions to donation. If a donation is inserted with a monthly or yearly recurrence, we will issue an insert into pledge command. When a recurring donation is automatically created with the previous trigger pledgesToDonations, this trigger will also activate. Thus, the interaction between recurringDonations and pledgesToDonations causes monthly donations to be generated automatically every month (and yearly every year) by simply updating the current date.

```
CREATE TRIGGER recurringDonations on Donation
    AFTER INSERT
    as
    BEGIN

    INSERT INTO Pledge
    SELECT donor_id, nonprofit_id, dateadd(month, 1, donation_date), amount,
recurrence
    FROM Inserted
    WHERE(recurrence = 'monthly')

    INSERT INTO Pledge
    SELECT donor_id, nonprofit_id, dateadd(year, 1, donation_date), amount, recurrence
    FROM Inserted
    WHERE(recurrence = 'yearly')

    END
```

However, something to note is that if someone enters a monthly donation from more than a month ago, the dateadd(month, 1, donation_date) would actually cause the inserted pledge to be in the past, which would create an error. We could address this by creating further triggers so that adding monthly donations in the far past would cause the months between then and the current date to be populated with monthly donations, but this seems forceful. It could also be an issue to auto-generate hundreds of donations if they enter a monthly donation from say, 20 years ago. Thus, we just raise

an error if they try to enter a monthly donation from more than a month ago or a yearly donation from more than a year ago.

7. *A monthly donation cannot be inserted where it's date is more than a month before the current date. Similarly, a yearly donation cannot be inserted from more than a year before the current date.*

Because a trigger created for this constraint would also be activated when inserting into donations, we can combine it with the previous trigger. Thus, we add to recurringDonations lines that throw an error if the user tries to enter recurring donations from too long ago. The final recurringDonations is:

```
CREATE TRIGGER recurringDonations on Donation
    AFTER INSERT
    as
    BEGIN
    IF EXISTS (SELECT *
                FROM INSERTED
                WHERE recurrence = 'monthly' and dateadd(month, 1,
donation_date)<=(select * from currentdate))
    throw 90000, 'You cant insert a monthly donation more than a month before the
current date', 1;
    IF EXISTS (SELECT *
                FROM INSERTED
                WHERE recurrence = 'yearly' and dateadd(year, 1,
donation_date)<=(select * from currentdate))
    throw 90000, 'You cant insert a yearly donation more than a year before the
current date', 1;

    INSERT INTO Pledge
    SELECT donor_id, nonprofit_id, dateadd(month, 1, donation_date), amount,
recurrence
    FROM Inserted
    WHERE(recurrence = 'monthly')

    INSERT INTO Pledge
    SELECT donor_id, nonprofit_id, dateadd(year, 1, donation_date), amount, recurrence
    FROM Inserted
    WHERE(recurrence = 'yearly')
```

```
END
```

8. *CurrentDate cannot change by more than a month at a time.*

   Aside from it being good practice to change the currentDate frequently, changing it by two months or more will cause issues with creating monthly donations. Thus, we create a trigger to prevent large changes to currentDate:

```
create trigger checkDateChange on currentdate
after update
as
if(abs(datediff(month, (select date from inserted), (select date from
deleted))) > 1)
 throw 100000, 'You cant change the current date by more than a month at a
time', 1;
```

9. *A single donation cannot have a dollar amount greater than 1 million dollars.*

This integrity constraint can be enforced with a simple check that can be added to bot the pledge and donation relations:

```
amount        numeric(7, 2), check(amount <= 1000000)
```

10. *The combined salaries of employees in a department cannot be greater than the budget of that department.*

Salaries are the most predictable expense, and the department will be incurring many expenses other than just salaries. So if this constraint is not met, there was not a legitimate attempt to fit the budget.

We can enforce this constraint by adding a trigger on insertion to employees and to budgets.

```
create trigger checkDeptBudget on budget
after insert
as
if exists(select *
          from inserted i
```

```
          where(select sum(salary)
                from employee e
                 where e.dept_name = i.dept_name and e.nonprofit_id = i.nonprofit_id) >
i.amount)
throw 110000, 'The sum of employee salaries cant be bigger than a dept budget', 1;


create trigger checkSalary on employee
after insert
as
if exists(select *
          from budget b
          where(select sum(salary)
                from employee e
                 where e.dept_name = b.dept_name and e.nonprofit_id = b.nonprofit_id) >
b.amount)
throw 110000, 'The sum of employee salaries cant be bigger than a dept budget', 1;
```

11. *No entry for "volunteer hours" can be greater than 24 hours.*

This is simply because there are only 24 hours in a day. This is enforced with a simple check statement:

num_hours NUMERIC(2, 0) NOT NULL CHECK(num_hours < 24)

8. Relational Database Design - Applying the Dependency Theory

A functional dependency {X → Y} for a relation R indicates that whenever two tuples in R share the same value for their X attribute, their Y attribute will also be the same. Relations that have functional dependencies risk storing redundant data. As an example, suppose that in our schema (though this is not true) all employees in the same department received the same salary. Therefore, {dept_name, nonprofit_id → salary}. Thus, having multiple employees in the same department would store redundant data about salaries.

However, these redundancies do not occur in our schema because they are in BCNF form. A relation is in BCNF form if for all functional dependencies {X → Y}, either it is a trivial dependency (meaning Y is a subset of X) or X is a key of that

relation. This prevents redundancies because by definition, if X is a key no two tuples will have the identical X values.

The functional dependencies for all relations are shown below. As seen, the left hand side of each of our functional dependencies is also underlined above, meaning they are all keys. Thus, all of the relations below are in BCNF form.

Nonprofit(nonprofit_id, npo_name, npo_address, npo_email)
{nonprofit_id → npo_name, npo_address, npo_email}

Donor (donor_id, donor_name, donor_gender, donor_address,
donor_phone_number donor_email)
{donor_id → donor_name, donor_gender, donor_address,
donor_phone_number donor_email}

Pledge(donor_id, nonprofit_id, pledge_date, amount, recurrence)

{donor_id, nonprofit_id, pledge_date → amount, recurrence}

Donation(donor_id, nonprofit_id, donation_date, amount, recurrence)

{donor_id, nonprofit_id, donation_date → amount, recurrence}

Employee(emp_id, dept_name, nonprofit_id, emp_name, emp_gender, emp_address, phone_number, emp_email, salary)

{emp_id → dept_name, nonprofit_id, emp_name, emp_gender, emp_address, phone_number, emp_email, salary}

Expense(exp_id, nonprofit_id, expense_date, amount)

{exp_id → nonprofit_id, expense_date, amount}

Department(dept_name, nonprofit_id, director)

{dept_name, nonprofit_id → director}

Budget(dept_name, nonprofit_id, year, amount)

{dept_name → nonprofit_id, year, amount}

Volunteer(vol_id, dept_name, nonprofit_id, vol_name, vol_gender, vol_address,
vol_phone_number, vol_email)

{vol_id → dept_name, nonprofit_id, vol_name, vol_gender, vol_address,

vol_phone_number, vol_email}

VolunteerHours(vol_id, nonprofit_id, volunteer_date, num_hours)

{vol_id, nonprofit_id, volunteer_date → num_hours}

Current_Date(date)
Because our relations are already in BCNF form, there is no need for any decomposition.

9. Revisiting the Relational Database Schema

There were some changes to our database as we started developing. For example, the "Budget" relation used to be an attribute of Department. However, we decided that it was more reflective of the real world if the budget was yearly. To maintain BCNF form, we needed to move budget into its own relation. Otherwise, a department instance would need to be created for each year it had a budget, which would needlessly repeat information such as director names.

However, the above sections already reflect our recent changes such as to budget. We believe that the relational model described in section 5 was sufficient for our queries and constraints. Therefore, our team agreed that we did not need to make any additional changes to our database here.

10. DBMS Implementation

As previously mentioned, we implemented our design in SQL Server, using VSCode and Microsoft Azure. The code base we used to create our database consists of two files: create.sql and DataGenerator.java.

create.sql consists of the list of CREATE TABLE SQL commands we used to generate the database structure, as well as CREATE TRIGGER commands to generate the triggers. DataGenerator consists of Java code that uses the SQL Server JDBC Driver to populate the database with sample data. This uses a series of files containing data for common first names (male/female), last names, and street names in order to procedurally generate donors, employees, and volunteers. The nonprofits were made manually, as we only wanted 9 of them for demonstration purposes, and because the naming conventions for a nonprofit are less conducive to procedural generation than the names of individual people.

A number of small challenges were encountered in this process, and several small changes were made. For example, expenses originally had a "type" attribute, but it was removed for being vague and difficult to define. Departments once had a "budget" attribute, but this was converted to a relation, due to the fact that budgets can often change year to year.

Some challenges were also encountered while generating data to populate the database. For example, generating random ID numbers that are also short is something of a challenge. The canonical way to generate unique ID's is with UUID's, but these are too long to be practically used by most systems (for reference, CWRU uses 7-digit numerical ID's). This was solved by inserting integers into a Set, which enforces uniqueness, before converting to a list. Other small challenges were things such as finding realistic ranges of salaries, donations, and budgets, as well as rounding values to have realistic values (it is unrealistic for people to donate $4378.62).

A few screenshots from our Azure interface are shown below to demonstrate our DBMS implementation.

(i) Showing our volunteers:

vol_id	dept_name	nonprofit_id	vol_name	vol_gender	vol_address
1049422	Outreach	2683272	Albert Sanchez	MALE	96168 61st Stree
1049692	Logistics	8612689	Carolyn Wood	FEMALE	60107 Water Stre
1050443	Logistics	3361510	Justin Thomas	MALE	14534 5th Street
1050492	Outreach	3361510	Teresa Hughes	FEMALE	5732 87th Place
1050939	Sales	2311476	Bobby Mendoza	MALE	50244 Madison L
1051518	Information Technol...	5273118	Gregory Carter	MALE	27315 Dogwood

(ii) Querying for the Red Cross's biggest donors for each month of the previous year:

```
2   SELECT month(D.donation_date), Donor.donor_name
3   FROM Donor, Donation D, Nonprofit, currentDate date
4   WHERE Donor.donor_id = D.donor_id
5   AND D.nonprofit_id = Nonprofit.nonprofit_id
6   AND Nonprofit.npo_name = 'American Red Cross'
7   AND abs(datediff(year, D.donation_date, date)) <= 1
8   AND D.amount = (SELECT MAX(D1.amount)
9   FROM Donation D1
10              WHERE month(D1.donation_date) = month(D.donation_date)
11          AND D1.nonprofit_id = D.nonprofit_id
12          AND year(D1.donation_date) = year(D.donation_date))
13
```

Results Messages

Search to filter items...

	donor_name
1	Judy Carter
2	Thomas Perez
5	Douglas Ramirez

(iii) Checking the “recurringDonations” trigger:

```
1  insert into donation
2  values(1048685, 3361510, '2/12/2021', 15000, 'yearly')
3
4  select * from pledge
```

Results Messages

donor_id	nonprofit_id	pledge_date	amount	recurrence
1048685	2258107	2021-05-17T00:00:00.00...	15000.00	monthly
1048685	2258107	2021-05-19T00:00:00.00...	15000.00	monthly
1048685	3361510	2022-02-12T00:00:00.00...	15000.00	yearly
1048685	8612689	2023-08-22T00:00:00.00...	77000.00	one-time
1048685	9927652	2022-06-10T00:00:00.00...	81000.00	one-time
1048685	9927652	2022-06-22T00:00:00.00...	55000.00	one-time

As seen, inserting a yearly donation for 2/12/2021 automatically creates a pledge for the next year, 2/12/2022 with all the other fields being identical.

The code for creating the relation schemas, enforcing the integrity constraints and populating the database can be found at https://github.com/menderbug/npoproject.

11. Application Implementation

We settled on a web application using Spring Boot to bootstrap a Restful R.E.S.T. (REpresentational State Transfer) Spring hibernate web application. The code base utilized to construct the webapp spanned approximately thirty (30) Java applications, between seven to nine java interfaces extending jpaRepository, and about a dozen static client side HTML5, JPA, and JavaScript files. We decided to use the Eclipse Java & Web Developers IDE in order to house all of our client side and server side files in the same development environment as our Tomcat Web deployment and Hibernate ORM applications. Our application implementation certainly leveraged Spring back end more than any front end development frameworks, although the DBMS was itself on the web.

One of the challenges we encountered in deploying this web application was the access kernel standardized across most major web browsers nowadays that prevents cross-site scripts. While this is certainly understandable from a security perspective, this constraint caused us quite the collective headache as we could not determine the best

course of action to circumvent this roadblock. Fortunately, the Spring framework as facilitated by Spring Boot via the web application Spring Initializr [sic], provided a workaround in the form of the springframework.cors.CorsConfiguration library, which allowed us to implement a @Bean annotation to filter out necessary transgressions against best practices. Since our Microsoft SQL Server was deployed on Microsoft's Azure cloud infrastructure, some degree of cross-site scripting was unavoidable, and the bean helped to redress this disparity between Browser default expectations and rules and crucial usage of API communications between the client side and server side database layer.

Further details on the web application itself may be found in the Appendices, which provide a much more detailed and pragmatic deep dive into understanding the webapp.

12. Conclusions

Overall, we felt that this project was fun and enlightening. I believe that doing this project has made the class concepts stick, and will make us more employable in the future. While we learned about making schemas, SQL queries and triggers in class, this project was really what put it all together. By doing this project, we were able to prove to ourselves that what we learned in class could have actual practical use in the real world.

Contributions:
David Ye: assisted in planning the structure of the database. Created a server/database instance in Microsoft Azure in which to implement the original concept. Ported initial concept code into SQL Server and used the associated JDBC driver to populate the server with sample data. In collaboration with Nick, iteratively improved and modified database structure from the initial concept.

Nick: Set up group meetings, assigned tasks to group members, wrote the integrity constraints and SQL queries.

Noel: Provided background information and did additional research on nonprofit organization structure, came up with entities, attributes, and relationships. Much of this was done through discussions with Victor.

Victor: Wrote portion of SQL Queries, Wrote Manuals, 1, 2, and part of 3. Built and deployed webapp using Spring Boot and Eclipse-maven-REST API

13. Appendix 1 - Installation Manual

The DBMS we used was Microsoft SQL Server hosted on Microsoft Azure. No installations were required/necessary for accessing the database itself, since it is implemented on cloud infrastructure, so only a Microsoft Azure account, a stable internet connection, and the necessary login credentials are necessary for querying the database directly. We developed the SQL Create commands and database population methods in Visual Studio Code, although any comparable IDE would work for the same purpose, and does not need to be downloaded and installed in order to operate the database. What is necessary, however, to have installed directly on one's computer, virtual development environment or otherwise, are the technologies used to construct the deployable web application. We used the Eclipse IDE for Java and Web Developers (2021-03) to implement and house all of the requisite deployment mechanisms, and the bulk of the web interface development was done using Eclipse, although eclipse based IDEs or frameworks such as Spring Tool Suite 3 will work just as well. The JRE used was Java SE 8 for the sake of simplicity, as the more sophisticated features in more recent versions and JDKs were deemed overcomplicating and superfluous for the needs of this web application. The client side (front end) static documents were uploaded to localhost port 8080 and port 3306 respectively using Apache Tomcat ver 9.0, and did not require a front end framework such as react.JS, Angular.JS, etc. just for instance. All static client side files are in plain HTML5, JavaScript or JSP packets, so no installations are needed to access the web based functionality for the application. However, for allowing the server side (Back end/API-service) Spring Boot application to communicate with the DBMS in SQL Server on Microsoft Azure, we used a combination of Hibernate and Spring Data JPA, both of which require download, installation, and their dependencies acknowledged in the project build documentation. To help simplify project dependencies, we built the web application using Maven build and repositories, which simplifies the transference and implementation, similar to how Gradle does, of the above technologies and plug-ins, namely of Tomcat ver 9.0, Hibernate 5.3, JPA 2.2, Spring Boot 2.4.6, and Java 8. When building the application, Maven compiles all of the requisite dependencies in a pom.XML file, which is reprinted below.

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.4.6-SNAPSHOT</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>nonprof.org.org.tiers</groupId>
<artifactId>nonprofit.dom.webapp</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>nonprofit.dom.webapp</name>
<description>Springboot Maven restful Hibernate
SQLServ</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
```

Web Operations

Get Nonprofit details
Update Nonprofit
Get Donor details
Update Donor
Get Employee details
Update Employee
Get Expense details
Update Expense
Get Department details
Update Department
Get Volunteer details
Update Volunteer
Get Volunteer Hours details
Get Nonprofit details

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web-services</artifactId>
</dependency>

<dependency>
<groupId>com.microsoft.sqlserver</groupId>
<artifactId>mssql-jdbc</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
<repositories>
<repository>
<id>spring-milestones</id>
<name>Spring Milestones</name>
<url>https://repo.spring.io/milestone</url>
<snapshots>
<enabled>false</enabled>
</snapshots>
</repository>
<repository>
<id>spring-snapshots</id>
<name>Spring Snapshots</name>
<url>https://repo.spring.io/snapshot</url>
```

```
Employee ID:
Nonprofit ID:
deptName:
Name:
Gender:
Address:
phoneNumber:
Email:
salary:
```

The complete pom.xml file is included with the application file zip folder, and this excerpt was included as an appendix to ensure that those installing the complete project have access to all of the dependencies for the project to function properly. The pom.XML (Project Object Model) is IDE agnostic, and can import the dependencies for the Maven project in most development environments assuming that the above technologies Tomcat ver 9.0, Spring tools 4, Hibernate 5.3, JPA 2.2, Spring Boot 2.4.6, and Java 8 are available project artifacts accessible through the environment's build functionality

14. Appendix 2 - Users Manual

In keeping with the nature of this project, the full functionality of the database is only really accessible from the Microsoft Azure DBMS. We felt that since the cloud framework was already web accessible, allowing not only for easy collaboration and development, but also to query the database via the web directly in the SQL Server DBMS, that the web UI/UX was best kept skeletonized as a proof of concept. In lieu of SQL injection parsers, firewalls, or other user data security mechanisms, the three tier architecture of the web application gets progressively more restrictive with the data access object(s). That being the case, the index.jsp landing page for the localhost deployed web application is at its core a list of hyperlinks, relating to "retrieve details" and "update__" functionalities respectively. Thus, the web application only implements the Read/Retrieve and Update portions of the C.R.U.D. Create, Read, Update, Delete Restful API framework for each of the respective seven primary relations leveraged from the SQL Server Database implementation. The rationale behind this design decision was not only that the true database administration should be done at a DBMS level, but also that Adding and Deleting entities are a rare occurrence for the specific use cases. New nonprofit organizations are not incorporated nearly often enough for

adding them via Http ${POST} operations, and by the nature of the database meant to house Nonprofit Organizations' records and information, deleting relation instances via a Http ${DELETE} operation proves even more few and far between. New people are introduced into and added to the nonprofit organization ecosystem fairly often, but can be done so just as quickly and easily from the SQL Server DBMS on Azure, and people rarely leave the ecosystem once they join because volunteers will likely not swear off helping out ever again, employees can not only go work for other nonprofit organizations in the database network but also boomerang back to original employers assuming they departed on amicable terms, donors that have not pledged to donate are still likely to donate to another nonprofit organization of their choice eventually, and all of the records of employment, volunteer hours and nonprofit patronage are stored within database for at least ten years, if not indefinitely. While Pledges and Donations are far more likely to iterate in and out of the database ecosystem on a regular basis, the web application does not include the relations Pledges, Donations or Budgets in the interest of preserving financial security. While nonprofits are required to disclose their financial statements to the government and the public, private donors are under no such obligation to release their past donation or future pledge details to the public domain. Thus, the web UI/UX consists primarily of a landing page listing hyperlinks to "retrieve details" of and "Update___" each of seven relations exposed to the internet via the server side API controller using Http ${GET} operations to read the JSON relation iterations, and using Http ${PUT} operations to update the entity attributes.

15. Appendix 3 - Programmer's Manual

Data Population Tool
The database is populated with DataGenerator.java. This populates the database with sample data when run. It contains the following classes and methods.
Classes:
NPO: contains fields to represent an NPO: ID, name, address, email, and wealth.
Person: contains fields to represent a Person: ID, first/last name, gender, email, phone number, address, and wealth.
Methods:
drop: drops all tables/triggers in the database. Used to reset the database when modifying the structure or data within.
populate: executes updates to the database to populate with all necessary data.
generateNPOs: creates 9 sample NPO's to demonstrate functionality.
generatePeople: procedurally generates any number of people (donors, employees, or volunteers).
randIDs: generates unique 8-digit numerical ID numbers.
randAddress: generates random street addresses.
randDate: generates a random date from 2000 to 2021.
futureDate: generates a random date from 2021 to 2023.
dateStr: converts a LocalDate to a formatted string.
randVal: gets a random value from a list.
rand: generates an integer within a range.

perturb: takes an initial value and generates a rounded value close to the initial value.

Web Application

This web application did not use a client side framework such as react.JS, Angular.JS, Bootstrap, CSS templates, etc. yet it did use the Spring 4 server side (back end API) framework and Spring Boot 2.4.6, so the majority of the functionality is on the server side implementation. As above, the server side codebase utilized a combination of Hibernate 5.3 and JPA 2.2 to query the Microsoft SQL Server database on Azure's cloud infrastructure as a part of its Spring Restful API. Hibernate is an object relational mapping library that translates between the relational database dialect for SQL Server using a JDBC driver and the Object Oriented plain old Java Objects (POJO) of server side entities. SQL Server uses the Create Table commands in Section 5 of this project report and java populating classes to construct and populate the database via DBMS, which is then connected remotely to server side Domain classes. Each Domain class (Stylized DOMAN in the file folders) contains a queryable java object with boilerplate constructors, getter, setter and toString methods that is mapped to the SQL Server database. Each Domain has a respective Repo (Repository) class that implements the jpaRepository interface that Spring uses to handle much of the operational logic of the server side back end webapp. Since the Repo class (aka Dao class) itself feeds much of the method headers from the extended parent class, the Repo body for each of the implemented relational tables only includes a single method fragment for finding the entity element by the ID attribute key. The Repo class(s) could easily have allowed searchable functions for each attribute of each relation, but we concluded that this was largely redundant and unneeded when using the unchanging ID key worked perfectly fine. The Repo Class for each relation is then injected into the Service class, which handles transactions across the server side application by feeding Spring jpaRepository operations to the API Layer exposed to the world wide web. The Service class keeps a repository object as a private field, and feeds find, retrieve and update commands upstream to provide data encapsulation and abstraction. This Service class for each relation is then injected into the Controller class (also called the Resources class), which converts basic java functions into Http ${GET}, ${PUT}, ${POST}, and ${PATCH} operations to send via AJAX to the client side application. Using web binding annotations to request and retrieve Http mappings, The Controller class keeps a service object as a private field and responds to Asynchronous JSON and XML calls (AJAX) from the client side, and returns JSON Strings in the form of read and update protocols mapped to ${GET} and ${PUT} commands respectively. Thus the server side application implements a RESTful rather than SOAP API schema, and serves Jquery and JPA calls from the front end client to the back end codebase, through each of the controller, service, repo, and domain java classes, through hibernate to the SQL Server DBMS, then takes the deliverable table, sends it back through the database, Dao, service, and resource layers to the client via parsed JSON (JavaScript Object Notation) objects deployed on Tomcat 9. This creates a closed loop of access with limitations for the UI/UX JSA and HTML5 files to the SQL Server DBMS, comprising the webapp functionality and operability.

private static String server =
"jdbc:sqlserver://npoproject.database.windows.net:1433;database=npoproject;user=azureuser@npoproject;password=6BILLIONturtles;encrypt=true;trustServerCertificate=false;hostNameInCertificate=*.database.windows.net

To access the database:

Credentials needed to access the SQL Server database:
Username: azureuser
Password: 6BILLIONturtles

This can be accessed online through the Azure Portal (https://azure.microsoft.com/en-us/features/azure-portal/), where our database has been shared with gxo3@case.edu.

Alternatively, the database can be accessed through VSCode, with the SQL Server extension. The process is this: in the SQL Server tab, add a connection. Enter the following information.
Server name: **npoproject.database.windows.net**
Database name: **npoproject**
Authentication type: **SQL Login**
Username: **azureuser**
Password: **6BILLIONturtles**
(Save password and display name are optional).