SNACKS

Sensor Network Around Charlottetown's Key Surroundings

Project Plan
Jan 2019

Jeremy Thompson
R.J. Arsenault
Eduardo Egger
Alexander Metcalfe

Vision Statement

This project consists of integrating The Things Network with a relational database from Amazon Web Services, designing a visualization for location-based environmental data, and producing documentation that encourages citizens to participate in the system (a low barrier to entry, low confusion about the system, and ease of access to information).

This project is being completed for the City of Charlottetown, but is ultimately being created with the goal of being a citizen-focused system. Citizens of Charlottetown are the primary users of this system, and primarily those among them that are scientifically minded or want to contribute to a citizen science project.

This project is being created with the hope of inspiring future development and engagement from citizens. Engaging citizens with a local project that uses novel technologies in a relatively new and exciting way is a great way to bring the community together, and a well implemented solution would be a major help to other communities worldwide who might be interested in a similar system. Another main goal of this project is to be able to use the recorded data to find irregularities in certain zones so that they can be addressed. This project was inspired by a very similar one that was made in Amersfoort.

Scope

The highest overview of the system is as follows. There will exist sensors scattered throughout Charlottetown, but near enough to a Things Network Gateway so that their data can be properly received. These sensors (most likely Arduino boards with a temperature sensor attached) will transmit their data at set intervals to The Things Network. Our system will take the data from The Things Network, associate it with geographic information we have previously gathered about the sensor, and store it in a more permanent database hosted by Amazon Web Services. Our project will also produce a web server that provides a public API for data retrieval from this database. Finally, as a demonstration of the API, a website that serves as the homepage for the project will display a map of Charlottetown with all of the sensors currently in use. The data from the sensors will also be visualized in some way (ex. temperature can be visualized with a colored heat map). Finally, there will be accessible documentation created that will also be available through the website that provides instructions on how citizens can get their own sensor from the city, how they can build their own sensor and register it with the city, or how they can remove an existing sensor from the system.

The data transfer from The Things Network to the Amazon database is our first priority. Once this is accomplished, creating the API that retrieves data from this database is the next step. Finally, the quality of the project's main website will depend on how well we adhere to the schedule, and a high quality data visualization can be considered a stretch goal.

Project Community

The project community will contain the four students continuing on from 4810 to 4820. Jeremy Thompson will be the project lead and R.J. Arsenault will be the technical lead. The two other developers are Eduardo Egger and Alec Metcalfe.

The primary customer is the City of Charlottetown. Our main contact with them (and the initiator of this project idea) is Peter Rukavina. Peter will also be able to maintain the system after the students graduate. Ramona Doyle is the Sustainability Officer at the City of Charlottetown, and Matt McCarville is the Community Energy Planner at the City of Charlottetown. They both have provided requirements for the system and what they envision it accomplishing in the future. There are also a number of interested citizens that Peter Rukavina has previously contacted, and they include Dave Cairns, Rosie Le Faive (from the Robertson Library), and Michelle Cottreau. These citizens are available as potential testers and can also help with specific technical areas of the project where they have expertise.

David LeBlanc will assist with the project planning aspects. He will help ensure that the proper steps are followed during the software development process, but will not assist in any specific technical aspect of the project.

Key Technical and Operational Requirements

This project is going to need technical aspects from PHP, MySQL, Amazon Web Services, Arduino, Data Visualization and Git. We will be developing the server side of the website using PHP, and the client side of the website using JavaScript. The database will be coded in MySQL, and be hosted using Amazon Web Services. We will require some knowledge on networking systems together since we are working with The Things Network and our own privately hosted database. On the website we will be using some data visualization tools such as D3 in order to present the data that is in the database. The sensors that we are using are Arduino, so we will need to understand and program in C in order to make them function. We will be using Git to maintain our code, so knowledge of that is also important.

System Acceptance Tests

Since this project is very loose in requirements from the client, the only tests we need to succeed in are coding the back end of the website in PHP, and using MySQL on an AWS server. If the project does succeed, a public API would also be a required result. We will test the system to ensure there are no security holes, and all the data is safe in the database. We will also test with simulated sensor data to ensure that the database is working properly, and that the data visualization is getting the right data and presenting it correctly. Tests will be made on multiple browsers. Any use of the API must not affect the database at all. We will ensure that the citizen/user of the system can interact easily with it, and ensure errors are minimal but exist when needed.

Use Cases and Scenarios

Use Case Name: Add Sensor to the System

Actors:

- User
- Administrator
- Website
- Database
- Things Network
- Sensor

Preconditions:

- User has received a pre-configured sensor from the City of Charlottetown that has been already registered in the Things Network application
- Or user has built their own sensor that they want to contribute data to the network

Post-conditions:

- The sensor will be able to write data to the main database
- The sensor will appear on the map that visualizes all sensors in Charlottetown

Normal Flow:

1. The user navigates to the website
2. The user selects the option from a menu to indicate they want to add a sensor to the system
3. The user is prompted to select the location of the sensor from an interactive map
4. The user is also asked to provide the unique identifying number that is on their sensor
5. The user is also asked to provide an optional nickname for this sensor
6. The information gathered from the user is submitted to the maintainers of the system
7. All this user entered data becomes associated with their specific sensor, and any future data provided by that sensor that goes into the database will be tagged with its geographic location data
8. The user provides power to the sensor and verifies that it has powered on properly
9. The sensor sends its first data value to the Things Network
10. The Things Network sends this data value to our database, which associates the unique ID of the sensor with geographic data that it received in step 3
11. The user verifies that their sensor appears on the map visualization

Exceptions:

3.1, 4.1, 5.1 User cancels the process and is returned to the website main page

Alternative Flow:

4.1 User does not have a unique identifying number, and selects that option

4.1.1 Confirm with use that they have not received this sensor from the City

4.1.2 Get information about type of sensor, type of data, and an email from user

4.1.3 Send this info to a system administrator who will review the request

4.1.4 If request is reasonable, administrator will add entry into Things Network for this citizen's sensor, and email citizen the sensor ID needed for Arduino code

4.1.5 If request is not reasonable, email citizen and let them know what is wrong

Use Case Name: Remove a Sensor

Actors:

- User
- Administrator
- Sensor
- Things Network

Preconditions:

- The sensor that is going to be removed was entered properly into the system previously

Post-conditions:

- The sensor will not longer send data to our database for storage

Normal Flow:

1. The user removes the power source from their sensor
2. The sensor will no longer send any data to The Things Network, and therefore no more data from the sensor will be put in the database
3. The user navigates to the website
4. The user selects the option from a menu to indicate they want to remove a sensor from the system
5. The user enters the unique sensor ID, and why they are choosing to remove their sensor
6. This information is submitted to maintainers of the system, and the user receives instructions about where to give the sensor back to the City of Charlottetown

Use Case Name: View Current Data From All Sensors

Actors:

- User
- Website
- Database

Preconditions:

- There is relatively recent data from the sensors in the database (the cutoff for "recent" data will be determined at a later date)

Post-conditions:

- The map will show the most up to date data from all sensors

Normal Flow:

1. The user navigates to the website
2. As the website is loading, it pings the database (via a publicly available API) for the most up to date information about all of the sensors in Charlottetown
3. The website computes how much of the map of Charlottetown it needs to show in order to fit all of the relevant sensors into the screen
4. The website draws the default visualization with the data it received from the database

Alternative Flow:

4.1 The user wants to view some sort of visualization other than the default type
4.1.1 The user selects the visualization that they want to see from a list of options
4.1.2 The website updates its display (no new data needs to be retrieved from the database)

Exceptions:

2.1 The API returns an error code instead of the most up to date data
2.1.1 The website displays an error page instead of a visualization

Use Case Name: View Current Data From One Sensor

Actors:

- User
- Website
- Database

Preconditions:

- The sensor the user wants to see has successfully been added to the database and has previously sent data to the database

Post-conditions:

- The user will see the most recent value from the sensor that is stored in the database

Normal Flow:

1. The user navigates to the website
2. As the website is loading, it pings the database (via a publicly available API) for the most up to date information about all of the sensors in Charlottetown
3. The website computes how much of the map of Charlottetown it needs to show in order to fit all of the relevant sensors into the screen
4. The website draws the default visualization with the data it has from the default sensors
5. The user clicks which sensor they want to see specific sensor information from
6. There is a pop-up display box of some kind that shows the value of the most recent data point, and the time that this data was sent

Exceptions:

2.1 The API returns an error code instead of the most up to date data

2.1.1 The website displays an error page instead of a visualization

Use Case Name: View Past Data

Actors:

- User
- Website
- Database

Preconditions:

- There is data in the database from the time period the user is interested in

Post-conditions:

- The user will be presented with a visualization of sensor data from some time in the past

Normal Flow:

1. The user navigates to the website
2. The user selects the time period that they're interested in from a menu of some kind (default selected option should be to display only the most up to date data)
3. The relevant information is retrieved from the database
4. The website updates to show the state of the data at the start of the user selected time period
5. The user can advance the state of the data by some amount (by one hour, by X hours, by X days, by X weeks)
6. As the time selection advances, the website will need to retrieve more data from the database to maintain smooth transitions
7. The website visualization will transition between different times by fading between colors or by some other kind of smooth transition, based on whatever kind of visualization is being displayed

Exceptions:

2.1 The user enters a time period in the future or that is otherwise invalid

2.1.1 The user is re-prompted to select a correct time

3.1 There is no data available for the selected time period

3.1.1 The API returns an error code to indicate no data was found for that period

3.1.2 The website does not update visualization, but displays a message that no data is available for that time period

Use Case Name: Sensor Sends Data

Actors:

- Sensor
- Things Network
- Database
- Website

Preconditions:

- The sensor and Things Network are both functioning properly
- The sensor is in range of a Things Network Gateway

Post-conditions:

- The database has a new entry from the sensor

Normal Flow:

1. After some set amount of time since its last data update, the sensor will have new data to send
2. The sensor broadcasts this data over LoRaWAN
3. The nearest Things Network gateway will receive this transmission
4. Our database will update with this new entry
5. The website will update in real time to reflect this new data entry with a smooth transition between the previous value and the new value

Use Case Name: Raw Data Access

Actors:

- User
- Database

Preconditions:

- The user knows how to work with the database API
- The user knows what specific information they want from the database

Post-conditions:

- The user will have the data that they requested from the database via the API in some raw, textual format (likely in the form of JSON)

Normal Flow:

1. User determines the specifics about what sort of data they want from the database (time span, data type, geographic bounds, etc.)
2. User sends the request to the database via an API
3. Database responds with the raw information that the user requested (likely will be in the form of a JSON formatted string)
4. User parses that information and does whatever they want with it

Exceptions:

3.1 User sends an improperly formatted request

3.1.1 Error code is returned instead of data

Feature List

Create Arduino code to gather and send data to The Things Network
Description: Arduino will be reading temperature data from a sensor. Need to code a way to read that data properly. Figure out a way to transmit data from the sensor to the Things Network. Decide which data needs to be sent to the Things Network.
Priority: 2
Effort: 1
Time: 20 hours

Test data integrity between Arduino, The Things Network, and the database
Description: Transmissions may not be received by the gateway for some reason. Ensure that this does not happen, or is at least minimized.
Priority: 1
Effort: 2
Time: 30 hours

Research data extraction from the Things Network
Description: The Things Network will store data for us but only for a small amount of time, so we need to find a way to move this data to a database.
Priority: 1
Effort: 2
Time: 6 hours

Create the data transfer scripts to send data from the Things Network to the database
Description: Create a solution that will transmit data from the Things Network to our database.
Priority: 1
Effort: 2
Time: 10 hours

Research data visualization techniques
Description: Need to find what data visualizations are available, and decide what data visualization to use.
Priority: 3
Effort: 2
Time: 14 hours

Build data visualization for the website
Description: Figure out implementation of the diagram, and where it will be processed (client side vs server side). Each point on the diagram will represent a sensor, and the area bounding the point will represent the area that sensor reading stands for.
Priority: 4
Effort: 4
Time: 40 hours

Decide what data we need to store in the database
Description: We need to figure out what data to get from the Things Network that needs to be stored in the database, and how to store it - what tables are required and which columns go in which tables as well as keys required.
Priority: 2
Effort: 3
Time: 15 hours

Learn and research AWS database
Description: We will be using Amazon Aurora for our database, need to figure out how it works.
Priority: 1
Effort: 2
Time: 10 hours

Design and implement database
Description: given all the decisions made about the database we must construct the database
Priority: 1
Effort: 4
Time: 16 hours

Research the AWS API
Description: Figure out how to use the AWS API.
Priority: 1
Effort: 3
Time: 10 hours

Develop the API
Description: Create functionality to allow polling the database for most recent data and old data, as the user desires. Also enable functionality to poll the database for a single sensors data.
Priority: 1
Effort: 3

Time: 20 hours

Design the front-end of the website
Description: The look and feel of the website, without the functionality
Priority: 2
Effort: 2
Time: 20 hours

Function to create csv files
Description: A function which takes all the data from the database and puts it in CSV format.
Priority: 2
Effort: 3
Time: 15 hours

Testing on API
Description: Need to be sure that the API will be secure, so no one can read/write data they aren't supposed to and no one can delete any data unless admin.
Priority: 2
Effort: 4
Time: 20 hours

Research Google Maps API
Description: Learn how to use Google Maps API
Priority: 3
Effort: 3
Time: 20 hours

Create documentation for end users
Description: people who have the sensors in their possession need to know how to care for it and maintain it
Priority: 2
Effort: 2
Time: 24 hours

Create documentation for API users
Description: users of the API need to know how to use it properly
Priority: 2
Effort: 2
Time: 12 hours

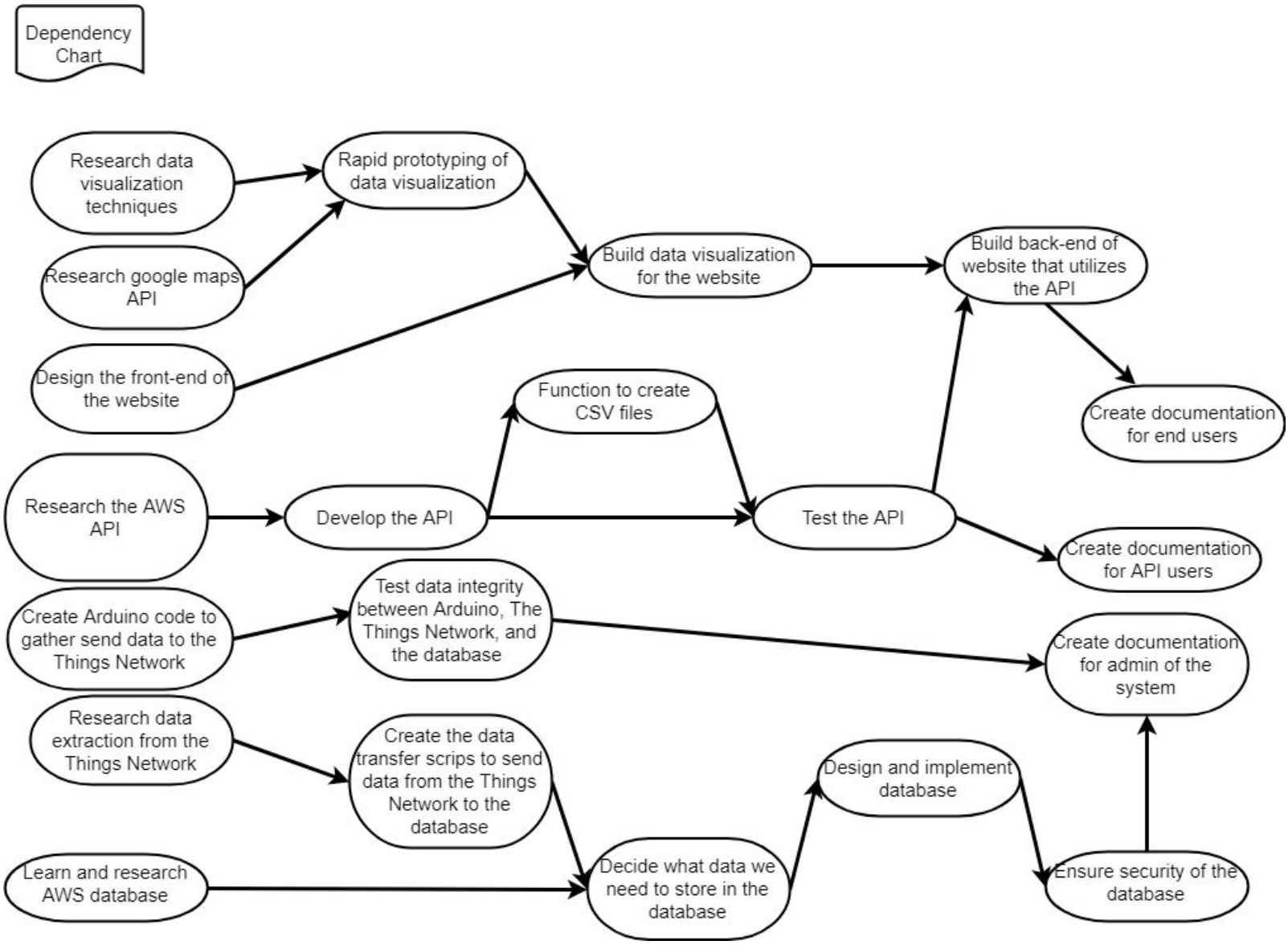
Create documentation for Admin of the system
Description: Admin needs documentation for how the system works
Priority: 2
Effort: 1
Time: 12

Ensure security of the database
Description: Ensure no undesired accesses or writes are happening
Priority: 2
Effort: 2
Time: 5

Rapid prototyping of data visualization
Description: Create a data visualization that satisfies the customers needs and looks good.
Priority: 3
Effort: 3
Time: 20

Build back-end of website that utilizes the API
Description: Create the functional part of the website that fetches data from the database using our API and presents it to a user in some format.
Priority: 3
Effort: 2
Time: 10

Dependency Chart



Delivery Plan

Sprint #1: January 7th to January 13th

- Work as a group to develop a project plan
- Ensure everyone is on the same page before starting the project

Deliverables:

- Project Plan that everyone on the team agrees on and understands

Milestones:

- Project started

Sprint #2: January 14th to January 27th - Researchability

- Create Arduino code to gather and send data to The Things Network (20)
- Design the front-end of the website (20)
- Learn and research AWS database (10)
- Decide what data we need to store in the database (15)
- Research data extraction for The Things Network (6)
- Research the AWS API (10)

Deliverables:

- A way to read sensor data in Arduino
- Table diagram of tables in database
- Sensor data can flow into The Things Network
- Method to extract data from The Things Network
- Website without data visualization
- Report on each option for the API implementation
- Report on AWS databases

Sprint #3: January 28th to February 10th - Workability

- Research data visualization techniques (14)
- Research Google Maps API (20)
- Design and implement the database (16)
- Create the data transfer scripts to send data from The Things Network to the database (10)
- Develop the API (20)
- Continued test of Arduino sensor and battery life (-)

Deliverables:

- Report on different visualization techniques
- Demonstration of Google Maps API
- Operational and documented database
- Data from The Things Network sending to database
- Functioning and documented API

Sprint #4: February 11th to February 24th - Testability

- Testing on API (20)
- Ensure security of the database (5)
- Rapid prototyping of data visualization (20)
- Build data visualization for the website (20)

Deliverables:

- A crude implementation of what the final data visualization will look like
- API tests

Milestones:

- Completed database
- Data travels reliably from Arduino to database

Sprint #5: February 25th to March 10rd

- Write public facing documentation for API (12)
- Build data visualization for the website (continued) (20)
- Test data integrity between Arduino and The Things Network and the database (30)
- Build back-end of website that utilizes the API (10)

Deliverables:

- Automated tests for the API, database, and data transfer

Milestone:

- Completion of the project

Sprint #6: March 11th to March 24th

- Function to create CSV files (15)
- Write documentation for Arduino usage (24)
- Write documentation for Admin to use the system (12)
- Start alpha tests on system with unfamiliar user (30)

Deliverables:

- Fully functioning API

- CSV files are being generated
- A large amount of documentation

Milestone: Complete API

Sprint #7: March 25th to April 7th - Clean up and bug fixing

- Clean up code and repository (30)
- Bug fixes as needed (20)
- Additional documentation as needed (20)
- Generate testing report (10)

Deliverables:

- Documentation
- Testing report
- Complete project

Milestone: Project completion

RMMM (Risk, Mitigation, Monitoring, Management) Plan

Risk: Title of sample risk
Mitigation: How can we avoid the risk? ***Monitoring:*** What factors can we track that will enable us to determine if the risk is becoming more or less likely? ***Management:*** What contingency plans do we have if the risk becomes a reality?

Risk: Lack of development experience Lack of experience with Amazon Relational Databases, PHP, and data retrieval API design could lead to overly confident time estimates and delay the project. Could also lead to poor quality code if systems are not fully understood.
Mitigation: Time will be spent in the initial stages of the project on learning each component. Each team member will familiarize themselves with each part of the system, before more specialization occurs during development. Peter Rukavina has extensive experience with Amazon Relational Databases and has offered help if we need it during this stage. Proper and thorough acceptance tests for each feature point will help mitigate this risk as well.

Monitoring: If development time is consistently longer than predicted for features that require these new learned technologies, the risk is more likely to be occurring. If developers attempt to use a technology but realize they will have to perform extensive research on it first, this indicates not enough time was allocated to learn the technology.
Management: If falling behind schedule, team members must then assess their ability to learn material on their own or ask for assistance from the tech lead or team lead, who will then potentially involve Peter.

Risk: Developer not putting in required effort
Whether it be because they are working on other class material or for whatever reason, a developer is not putting in the time and effort and the deliverables are suffering as a result.

Mitigation: Team leads (and developers themselves) should gauge how much they are able to realistically accomplish during a week, and only accept tasks they know they can handle. Proper motivation and planning contribute to this as well.
Monitoring: If a team member is consistently taking on less work the other members, or if they are consistently failing to meet the goals set by a specific feature, it is likely they are not putting forward the required effort.
Management: Ideally developers will recognize that they are not putting in enough effort and will self correct. If this does not happen, the team lead could have a conversation with them. If problem persists, Dr. LeBlanc will be alerted.

Risk: Amazon Relational Database Cost
The Amazon RDS costs more than we were expecting to use.

Mitigation: Proper study of the documentation of the Amazon RDS and gradual integration of the system will mitigate any changes that would cause a large amount of data to enter or be retrieved from the database. Incrementally changing the quantity or frequency of data retrieval will prevent major changes that would affect pricing.
Monitoring: A continual monitoring of the price charged to the credit card on the account will allow for detection of increased cost.
Management: A limitation on hourly or daily database access could be imposed by our API, and in the Things Network sensors code. If all else fails, the database can be temporarily stopped to fix the problem of increased cost.

Risk: Things Network unreliable The data transfer between individual sensors and the Things Network Gateway is not consistent and leads to missing or corrupt data.
Mitigation: Checking the data that is received for validity is a good first step. Testing of the sensors and their reliable working range would allow for rough estimates of how far from gateways individual sensors need to be. **Monitoring:** If sensors in the network suddenly go offline or start reporting data at irregular intervals. Or if sensors start reporting values that are inconsistent with reality or other sensors in close proximity. **Management:** Sensors must be placed closer to a Things Network Gateway. As this system is mostly a proof of concept, more gateways could be added to the system in the future, but a fully functional system that covers all of Charlottetown is out of the scope of this project.

Risk: The Things Network APIs are unstable
Mitigation: Testing of all APIs that we choose to use, and studying the quality of documentation beforehand, as well as possible levels of support **Monitoring:** If a large amount of time is spent trying to use an API in a way that we expect to work but does not perform as we expect, this risk is becoming more likely **Management:** We either have to use another API or change the logic of our approach. This system relies on The Things Network to work, so we have no choice about interacting with it.

Risk: Developer goes rogue A team member starts implementing things on their own without following the plan to a reasonable degree, and without informing the rest of the team until they've completed or failed their self assigned task.
Mitigation: Regular stand up meetings that developers commit to will ensure that everyone is aware of what everyone else is working on and that they are not deviating from the project plan in a major way. Ensuring everyone participates during the creation of the final project plan hopefully means that final decisions will be respected and followed through on by developers.

Monitoring: If team members under report what they are doing during stand up meetings, or if they have a great disagreement with the project plan, they are more likely to disregard it. Tracking progress of features will also reveal insights about this risk.
Management: Team lead can intervene and remind developer of the importance of the project plan. If ideas that developer is working are are an improvement over the original project plan, they can be integrated, but team lead must stress the importance of open communication between members, especially related to what each team member is working on at a specific moment in time.

Risk: Battery life of sensors
Battery that is used to power the Arduino device might die way faster than we anticipated.

Mitigation: Test the sensors before finalizing their version and code.
Monitoring: If the sensor dies or gives inconsistent readings while well in range of gateway, battery might be dying.
Management: Find a larger power source for the device. As a last resort, plug each sensor into a physical power outlet and only use a battery as a backup power supply.

Risk: Failure to meet schedule

Mitigation: Continual monitoring of past time estimates and how long tasks actually take will allow future time estimates to be adjusted appropriately. Developers should not strive for perfection in their tasks, but for "good enough".
Monitoring: The actual project progress should match with the proposed timeline, and if not it indicates a problem in either estimation or effort.
Management: Adjustment of future estimates. Encouraging harder work on current tasks would also help, but not to the point of extreme overtime hours.

Risk: Requirements not understood fully

Mitigation: Meetings with client at milestones will enable feedback to determine if we are on the right path and are continuing to build the correct product. If not, the iterative process can correct where we went wrong before proceeding

Monitoring: Feedback from clients is only way to determine if our product is what they are looking for or not.
Management: Re-iterate over requirements if client is not happy with the current state of the product. Ensure that any questions from developers are posed to the clients without delay, no hesitation in asking questions in a professional manner.

Risk: Tests are not written because of time limits

Mitigation: Plan adequate time during development for writing tests for individual components of the system, and do not leave all the testing until the end of the project. Determining an automatic testing suite to use (ex. Jenkins for integration with GitHub, or any other system)
Monitoring: During early phases of project, if tests are not being written, need to stop development and make sure that automated tests are written before proceeding with development
Management: Time needs to be made for tests. Developers will need to put in extra hours to write tests if the time crunch is really that drastic.

Risk: Writing tests takes too much time
Since writing comprehensive test suites for code is relatively new to all of the team members, time will have to be spent on learning how to properly set up an automated test environment and how to write tests for it.

Mitigation: Plan adequate time very early on in the process dedicated to learning about what Continuous Integration or Automated Testing platform we will be using, and how we will determine which tests pass or fail (automated open source system likely easiest for this).
Monitoring: If test writing goes over time estimates consistently, this risk is occuring
Management: Future estimates for testing could be slightly increased. In theory, as more tests are written developers should become more proficient at writing these tests and finish them quicker, so be reasonable with the amount of time added to future test writing.

Risk: Inadequate security

We do not want anybody to be able to modify the database unless they are a sensor in the Things Network, in that case we want them to be able to add new data. However, we need a data viewing API that will provide the functionality for our website.

Mitigation: Thorough testing of our API and Amazon RDS against access without credentials. Amazon likely has decent security as long as we set up out credentials properly, but the bigger risk is testing our API to make sure the only thing that it can do is view existing data.
Monitoring: The only factors that will provide any hint to a security risk is our tests. Overtesting things is not a bad idea here, since we want to cover as many possible security breaches as possible.
Management: Investigate how we can fix the issue. This might require reconfiguring our Amazon database or changing our API. Depending on the level of the security fault, the system might need to temporarily be disabled.

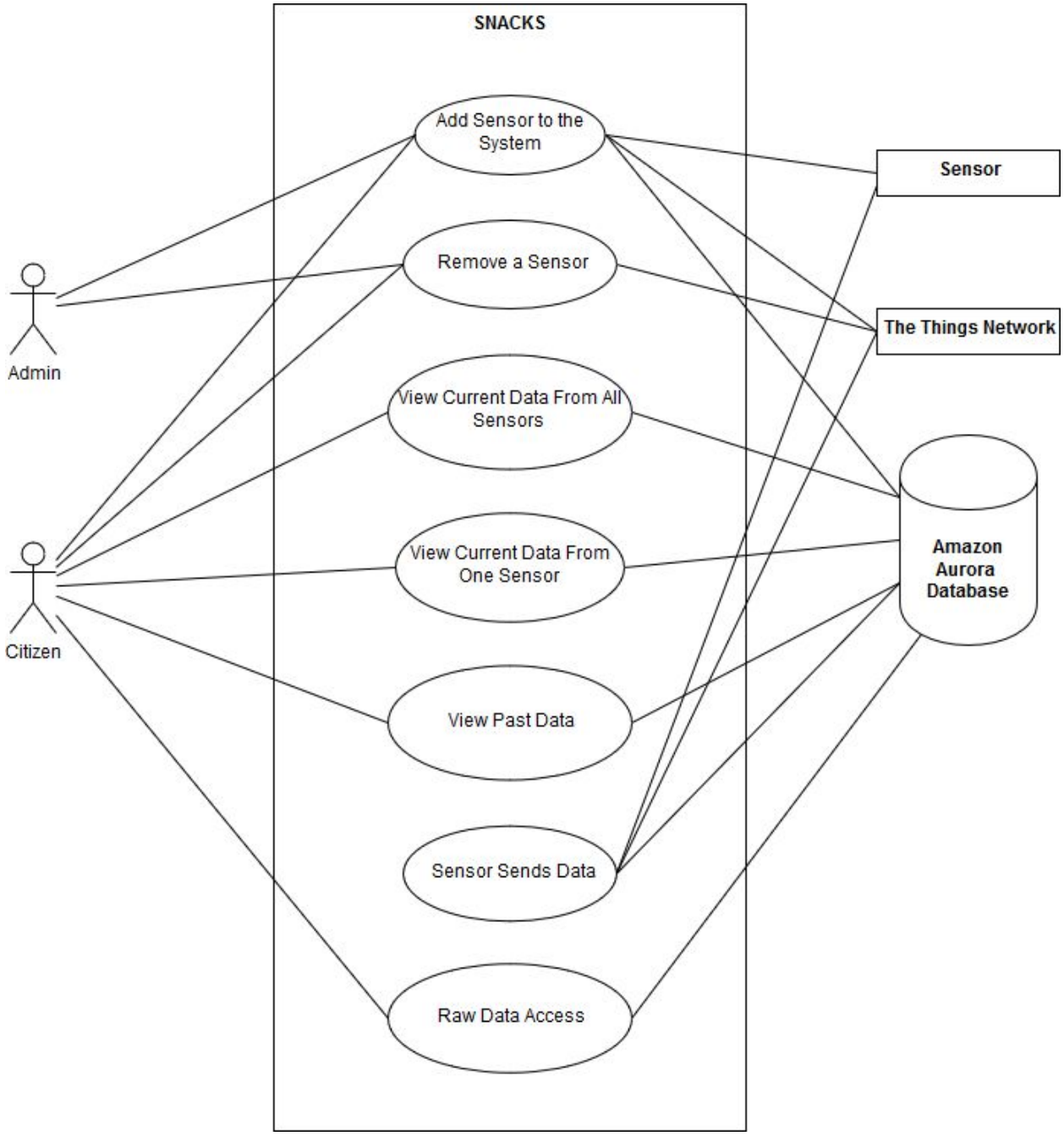
Risk: City uncertainty
Unclear requirements from the city lead to disappointment in the final product.

Mitigation: Throughout development of the project, any uncertainty needs to be presented to the city, which will lead to a concrete decision.
Monitoring: If developers are constantly asking questions about the project requirements, then this risk is becoming a reality.
Management: Schedule more regular meetings with the city or with Peter. Keep them informed of any current confusion we have about the project - these confusions should also come to light during the frequent stand up meetings between developers.

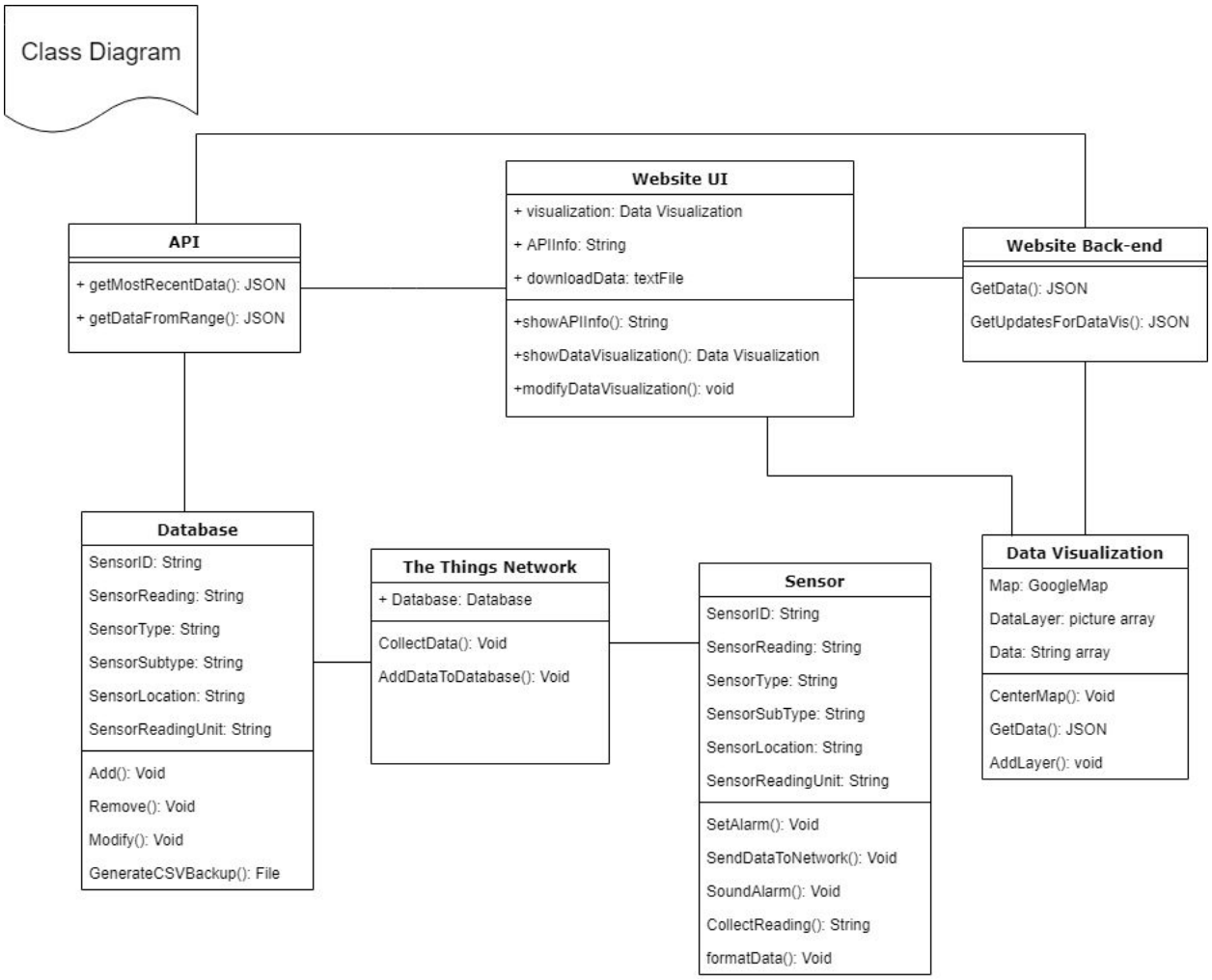
Risk: Environmental factors
Winter weather limits the amount of time that we can devote to the project.

Mitigation: Staying on top of work and getting work done when the weather permits it.
Monitoring: If a lot of time is missed due to weather then this risk is becoming real.
Management: If time missed is substantial, then weekly time worked per developer will have to be increased.

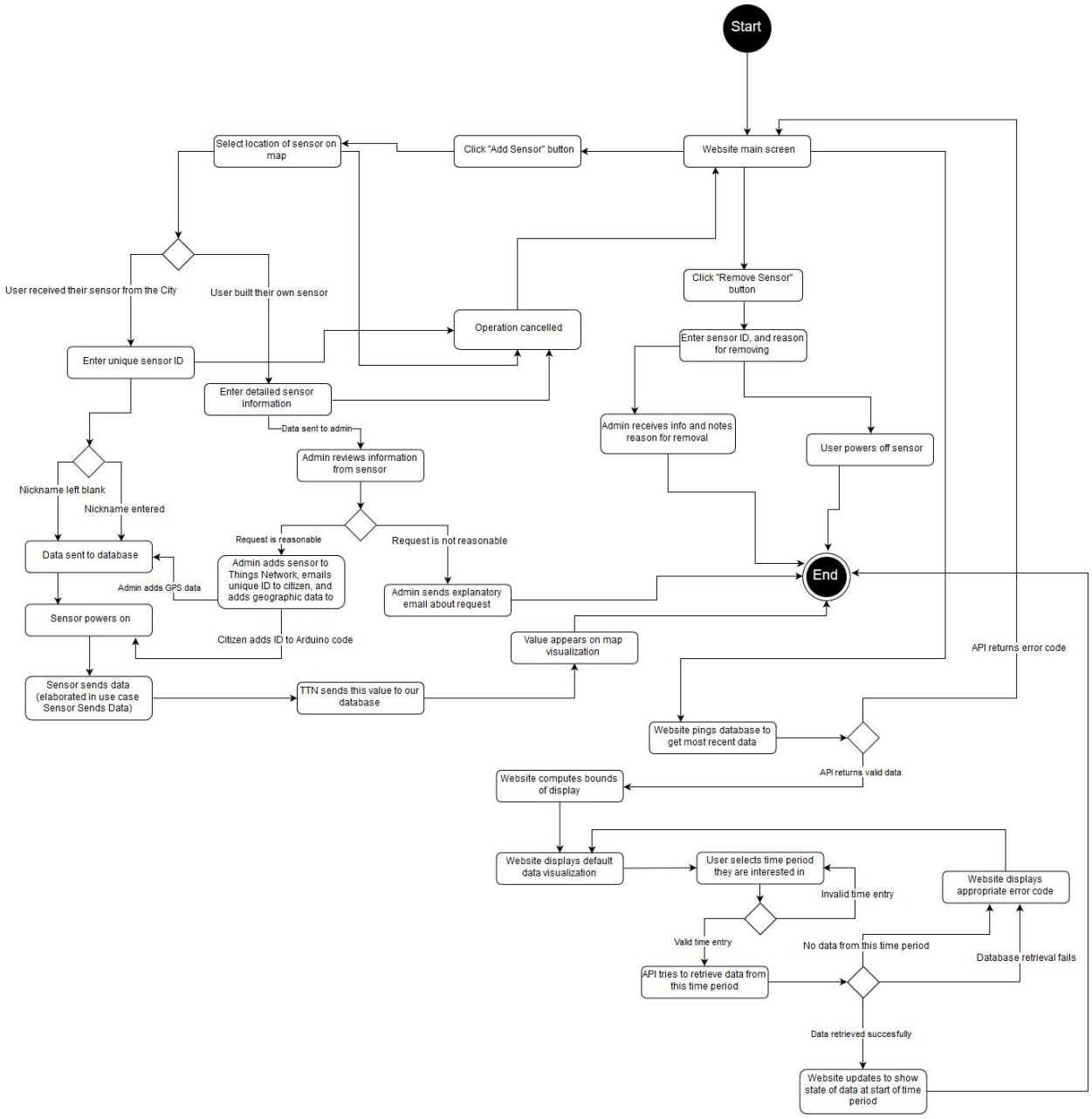
Use Case Diagram



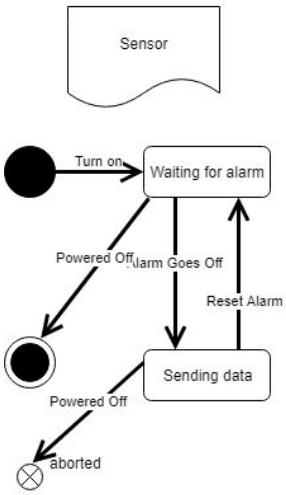
Initial Class Definitions

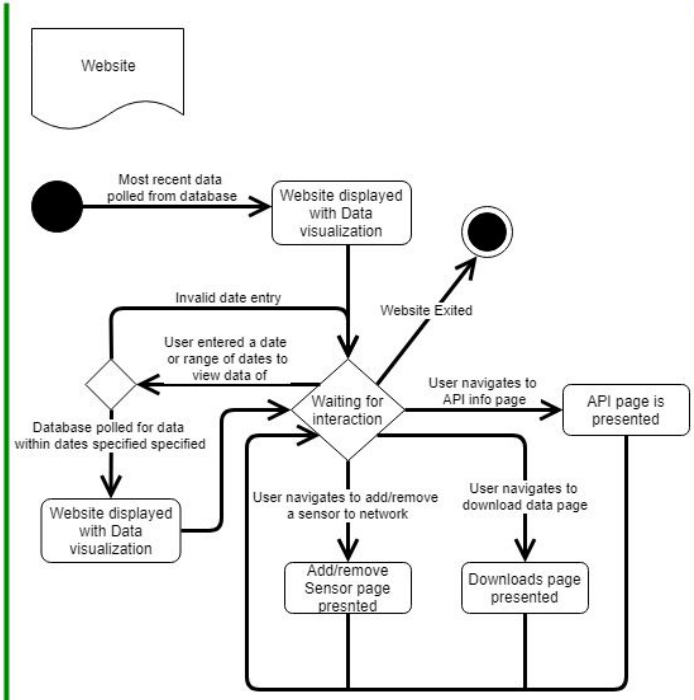


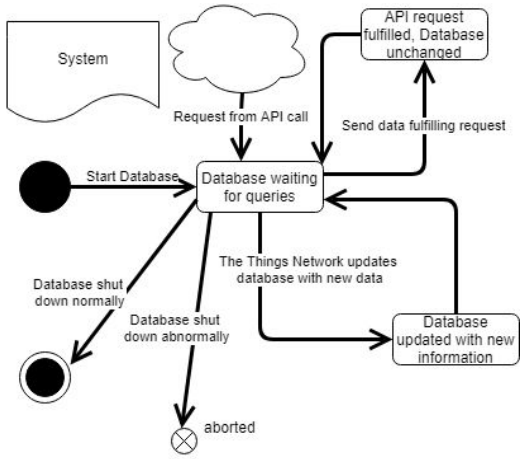
Activity Diagram



State Diagram







CRC Model

Class: Sensor	
Responsibilities: ● Collect data ● Format data ● Send data ● Sound Alarm ● Set Alarm	**Collaborations:** ● The Things Network

Class: The Things Network	
Responsibilities: ● Gather data from all sensor classes ● Store data in database	**Collaborations:** ● Sensor ● Database

Class: Database	
Responsibilities: ● Collect all the data ● Respond to queries ● Remove data ● Modify data ● Generates backups of data in CSV format	**Collaborations:** ● API ● The Things Network

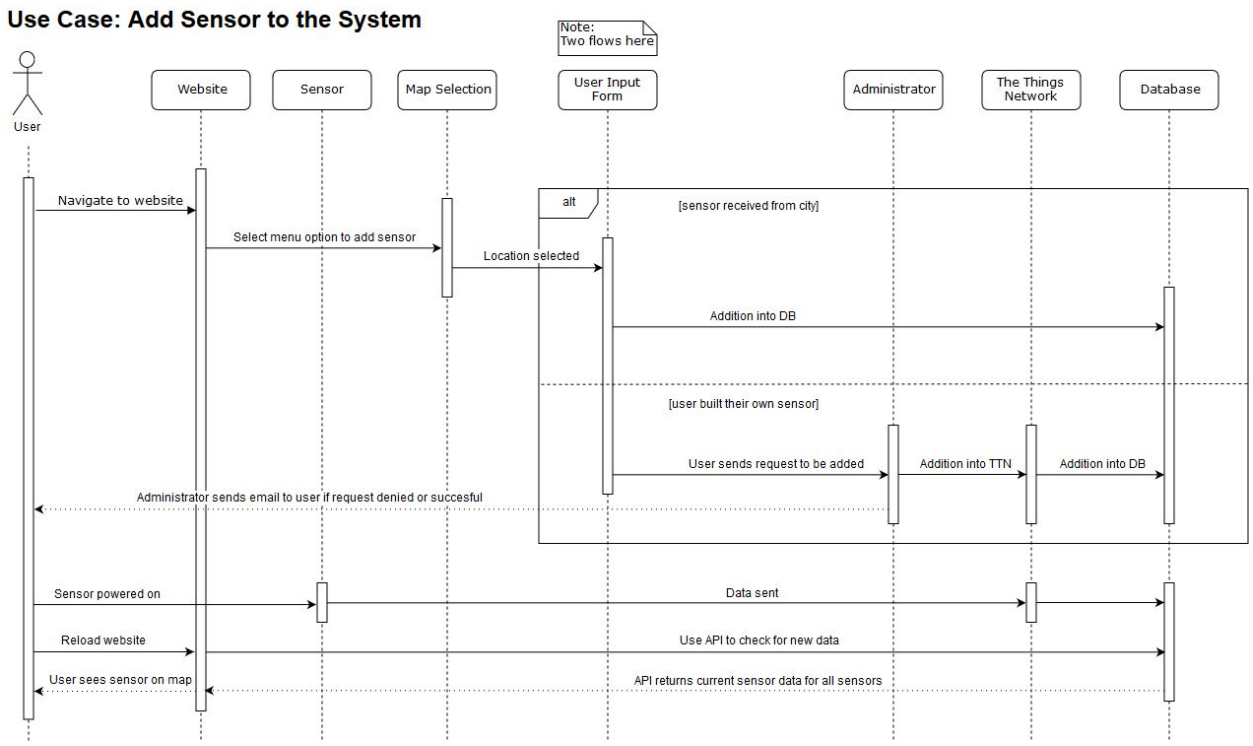
Class: API	
Responsibilities: ● Poll database for most recent data ● Poll database for some previous data	**Collaborations:** ● Database ● Website UI ● Applications from the general public

Class: Website UI	
Responsibilities: ● Show information on how to use the API ● Show Data Visualization ● Send modifications to the Data Visualization as the user wants to the website back-end	**Collaborations:** ● API ● Data Visualization ● Website Back-end

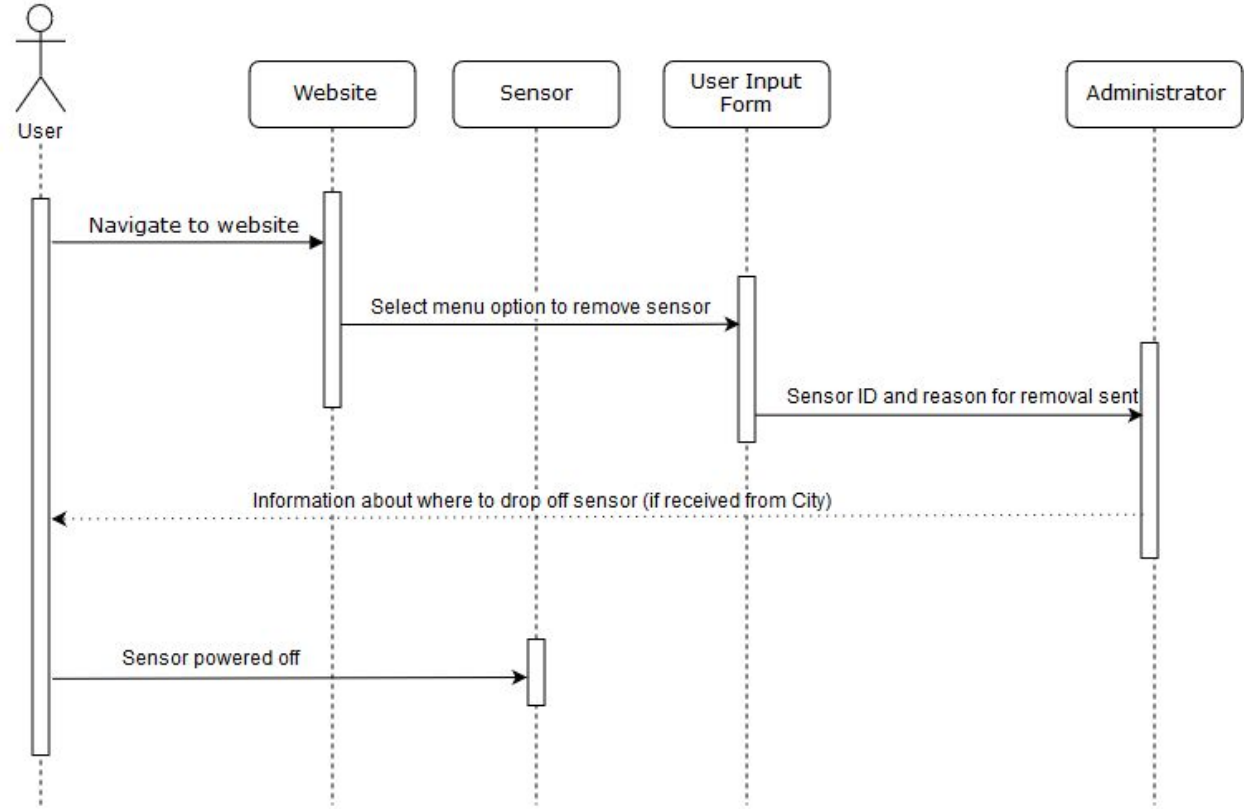
Class: Data Visualization	
Responsibilities: ● Center Google Map over Charlottetown ● Get most recent data from Website Back-end ● Add layer over map representing sensors ● Add layer over map representing area around sensors.	**Collaborations:** ● Website UI ● Website Back-end

Class: Website Back-end	
Responsibilities: ● Get most recent data from the API ● Communicate with Website UI for changes to the Data Visualization	**Collaborations:** ● API ● Website UI ● Data Visualization

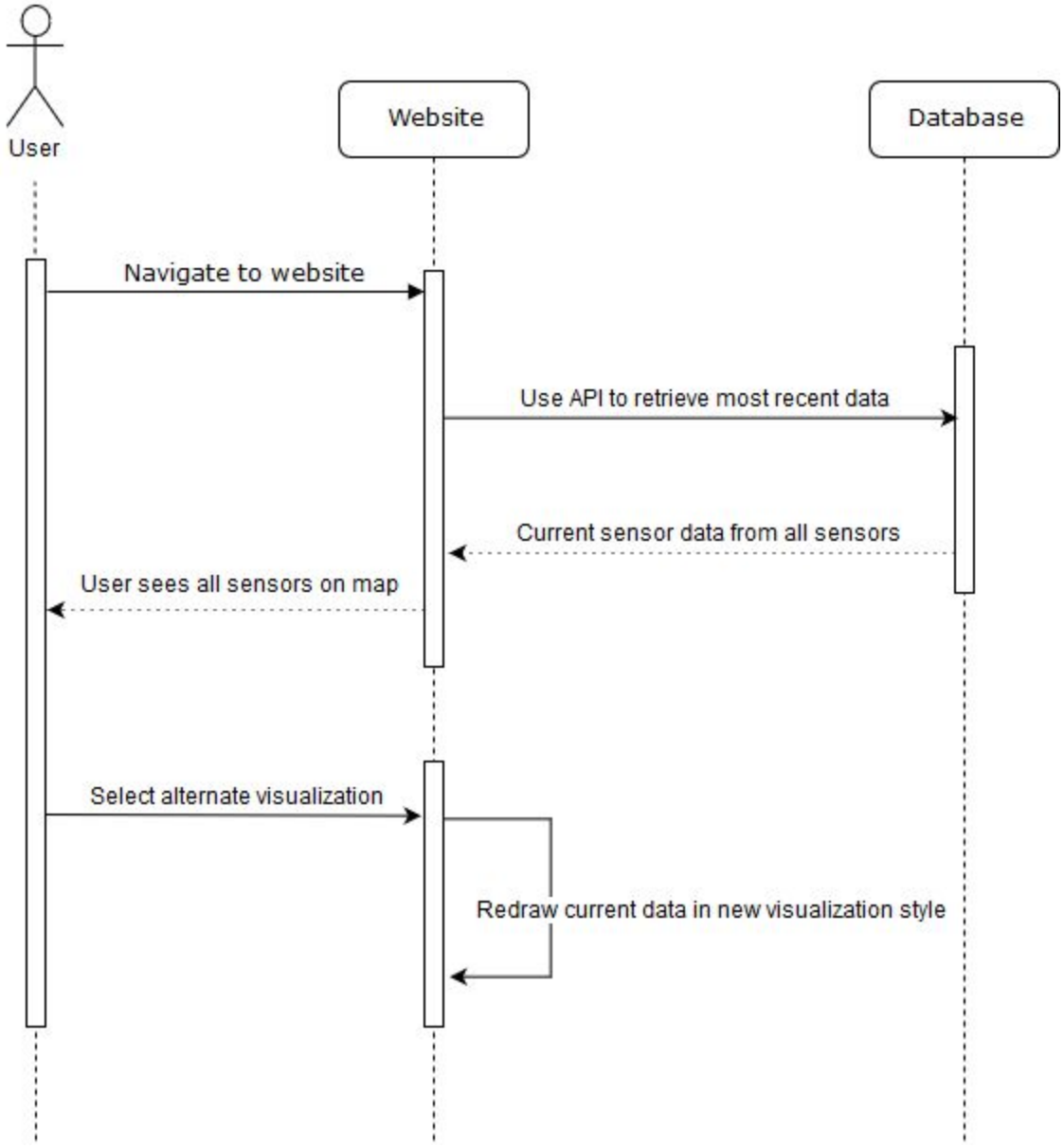
Sequence Diagrams



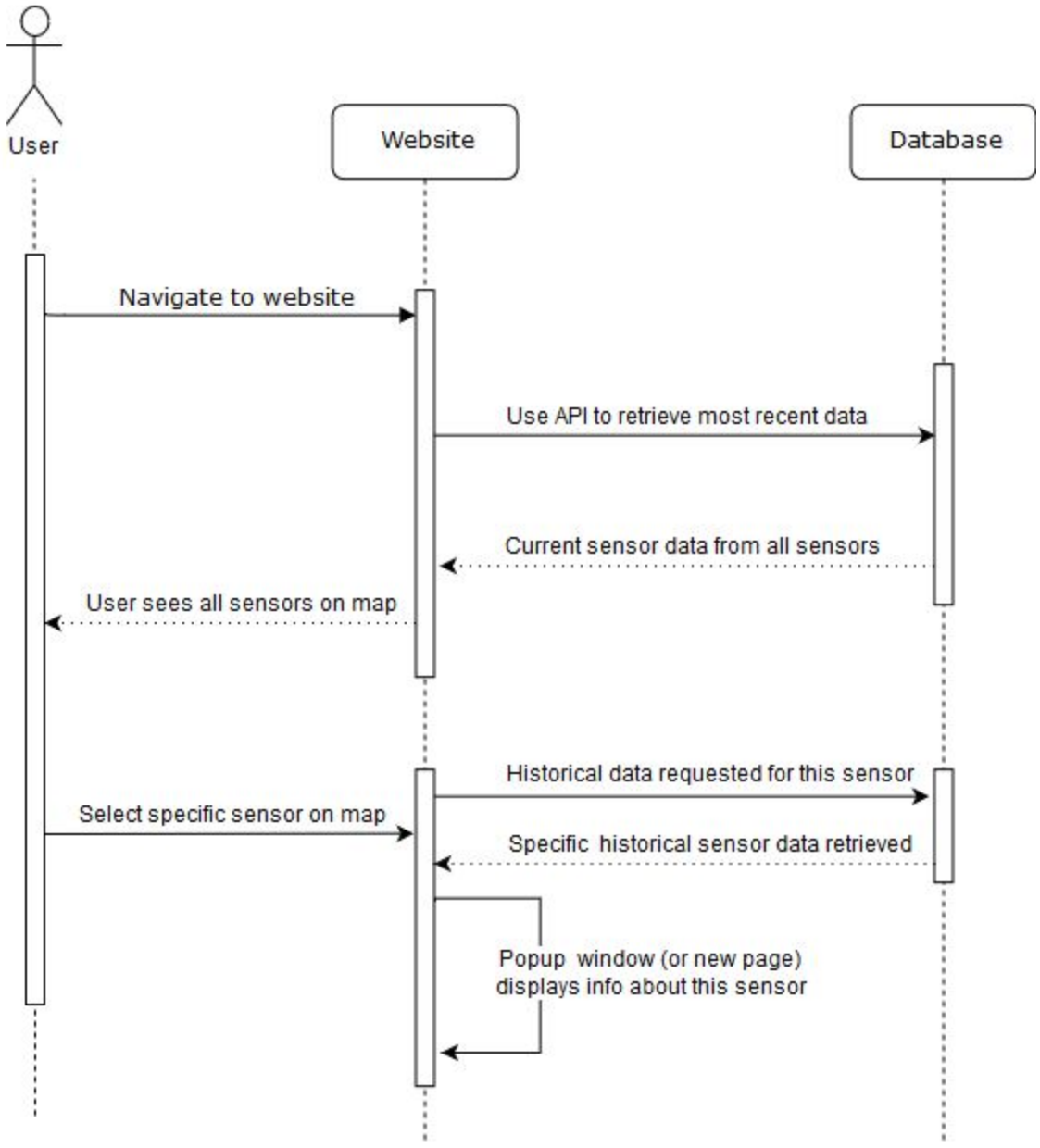
Use Case: Remove a Sensor



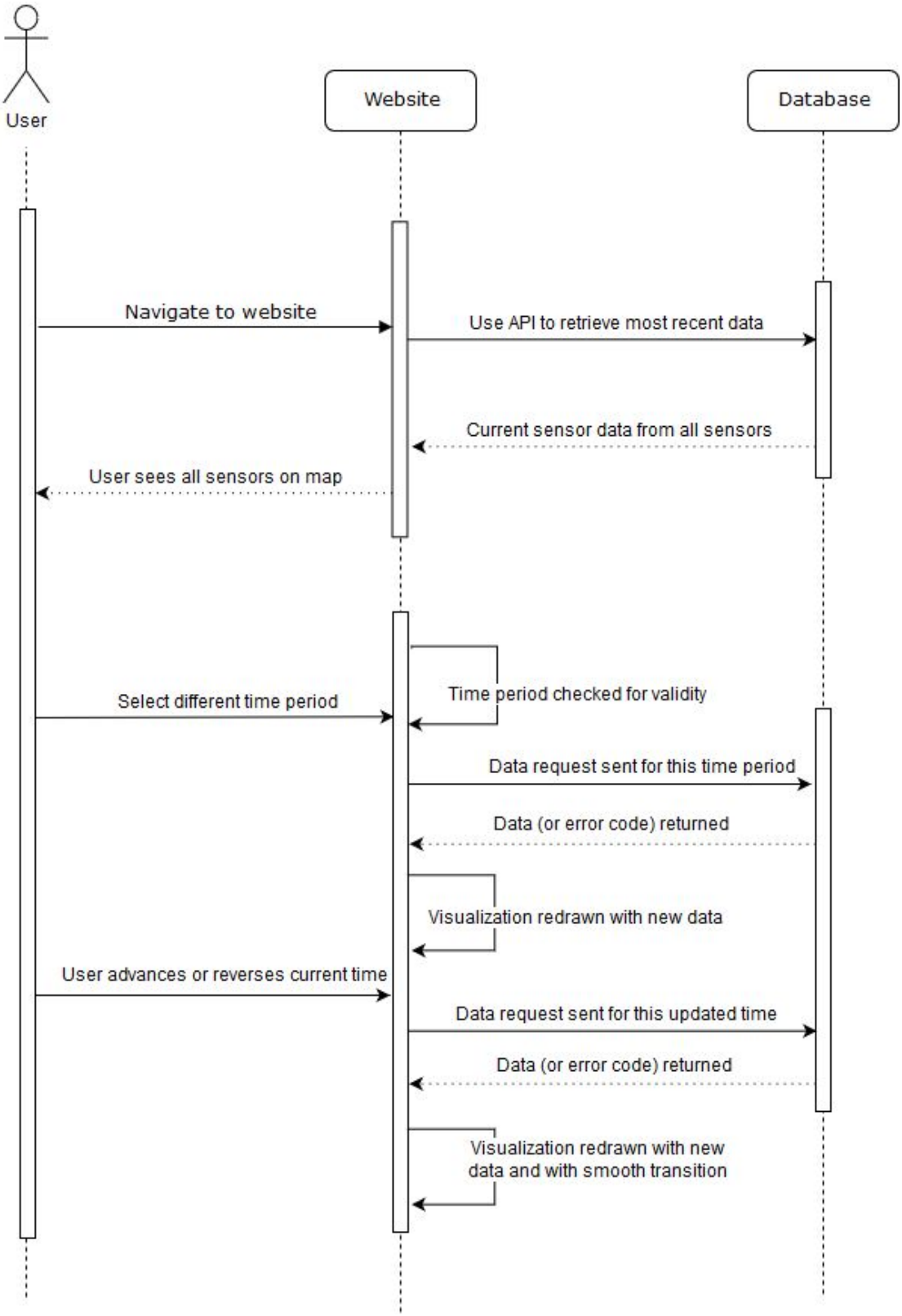
Use Case: View Current Data From All Sensors



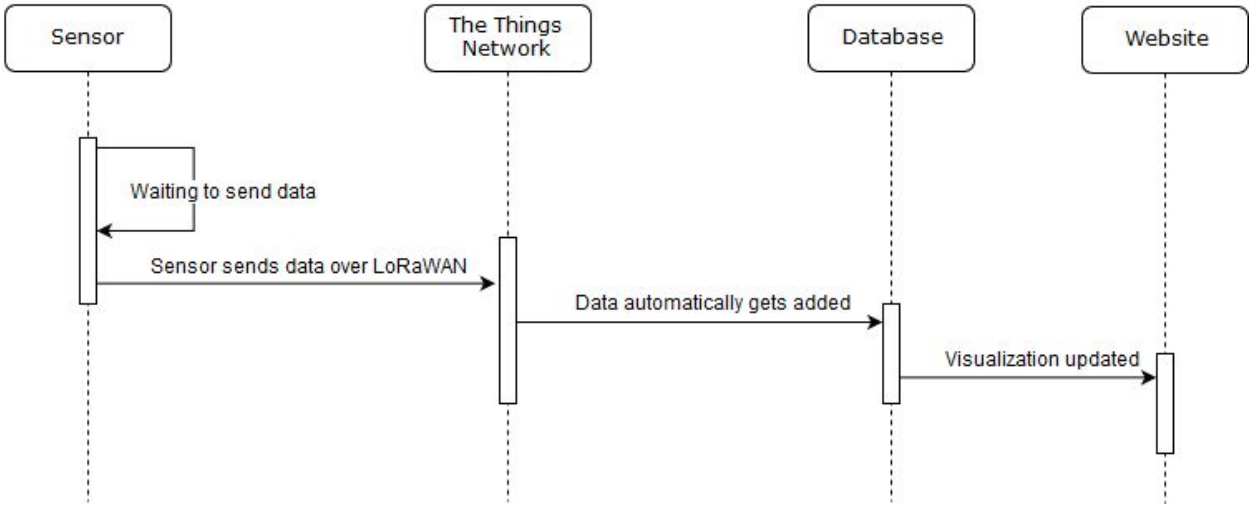
Use Case: View Current Data From One Sensor



Use Case: View Past Data



Use Case: Sensor Sends Data



Use Case: Raw Data Access