SRS Documentation for the Gridwatch System

Created by the Denied Access Team of the University of Pretoria COS 301 Module

- Anru Nel (u20646284)
- Joshua Young (u20442018)
- Leonardo Wessels (u17229457)
- Tshegofatsho Motlatle (u17066736)
- Thebe Kgaphola (u18371435)

Introduction

The vision of this project is to provide a customer relationship manager for a citizen who would like to report and receive feedback on a municipal issue that they are experiencing in their area. The goal is to provide an easy to follow, consistently updated, system that will serve as a way for a customer to keep track of their issues and a management system for city officials and technician teams trying to resolve the issue.

The need for this application is seen daily. With multiple reports of hundreds of “check-up” calls happening daily on our municipal grid (2000 Calls Linked to Power Outages, Electrical Outage Results in 800 calls, Power Outage Results in 3000 Calls) the need for a system which can streamline this process is almost essential. We intend to provide a ticketing system which allows for this check-up system with some automated features.

User Characteristics

1. Normal User
   This would define a typical everyday user of the GridWatch System. This user would

use the system to create an issue, edit an issue, check information about their issue and view issues in their area.

2. City Official
   This would be a member of a city municipality who would use the system to dispatch issues, sort through all issues based on their features and view all issues that are open

3. Technician Team
   This would be the group of people that receive dispatched issues from the City Officials. The tech teams would use this system to update progress issues, accept or deny new requests and provide other basic information about issues.

Functional Requirements

Requirements

1. GS shall allow a user to register an account with the GS
   1.1. GS shall provide a user the same issue registration functionality as a guest
   1.2. GS shall register tickets that are created by a user to that user
   1.3. GS shall allow a user to view updates on a user's issue
   1.4. GS shall keep a hidden rating of a user to measure the validity of their issue reports
2. GS shall allow a guest to register an issue
   2.1. GS shall allow a power failure to be registered
   2.2. GS shall allow a water supply failure to be registered
   2.3. GS shall allow a pothole to be registered
   2.4. GS shall allow a sinkhole to be registered
   2.5. GS shall allow a user to upload a picture with the issue they are registering
   2.6. GS shall provide a guest facility to provide more information about the issue being registered
      2.6.1. GS shall require a user to pin the location of the issue being registered
      2.6.2. GS shall require a user to take a picture of the issue being registered
      2.6.3. GS shall require a user provide a description of the issue being registered
3. GS shall provide a web-based portal (referred to as GSWP) for authorized officials
   3.1. GS shall display user created tickets to an authorized official

3.1.1. GS shall allow a user to sort displayed tickets

3.1.1.1. GS shall allow an official to sort tickets by upvotes

3.1.1.2. GS shall allow an official to sort tickets by ticket progress

3.1.1.3. GS shall allow an official to sort tickets by pending acceptance status

3.1.1.4. GS shall allow an official to sort tickets by ticket opened status

3.1.1.5. GS shall allow an official to sort tickets by ticket location

3.1.1.6. GS shall allow an official to sort tickets by the date the ticket was created

3.1.2. GS shall display average time to complete similar issues

3.1.3. GS shall allow a ticket to be explored

3.1.3.1. GS shall allow an official the option to dispatch a ticket to tech teams

3.1.3.1.1. GS shall ensure that tickets are automatically sent to relevant tech teams

3.1.3.2. GS shall allow an official to view details about a user created issue

3.1.3.3. GS shall allow an official to update the status of a ticket

4. GS shall communicate newly registered issues to authorized officials

4.1. GS shall create an entry for the registered issue on the GSWP

4.2 GS shall send a notification to authorized officials currently using the GS

5. GS shall provide a cell phone application for technicians

5.1. GS shall display tickets relevant to the technicians

5.1.1. GS shall allow a pending ticket to be explored

5.1.1.1 GS shall allow a pending ticket to be accepted or denied

5.1.1.1.1 GS shall, upon a ticket being accepted, assign the ticket to the accepting tech team

5.1.1.1.2 GS shall, upon a ticket being denied, mark the ticket as not suitable for the denying tech team.

5.1.2. GS shall allow an accepted ticket to be explored

5.1.2.1 GS shall allow a technician to communicate ticket details with the public

5.1.2.1. GS shall communicate to the public a ticket work start date

5.1.2.2. GS shall communicate to the public a ticket estimated repair cost

5.1.2.3. GS shall communicate to the public a ticket estimated repair time

5.1.2.2 GS shall allow a technician to update the status of a ticket

6. GS shall provide a cell phone application for the public

6.1. GS shall facilitate a user viewing all reported outages in their area

6.1.1. GS shall provide information about specific outages in a user’s area

6.1.1.1. GS shall provide the estimated progress of a ticket to the user

6.1.1.2. GS shall provide the estimated cost of a ticket to the user

6.1.1.3. GS shall provide the ticket start date to a user

6.1.2. GS shall allow users to up-vote tickets and for that tickets up-votes to be attached to the issue

REMAINING REQUIREMENTS AS AT DEMO 3

1.4.

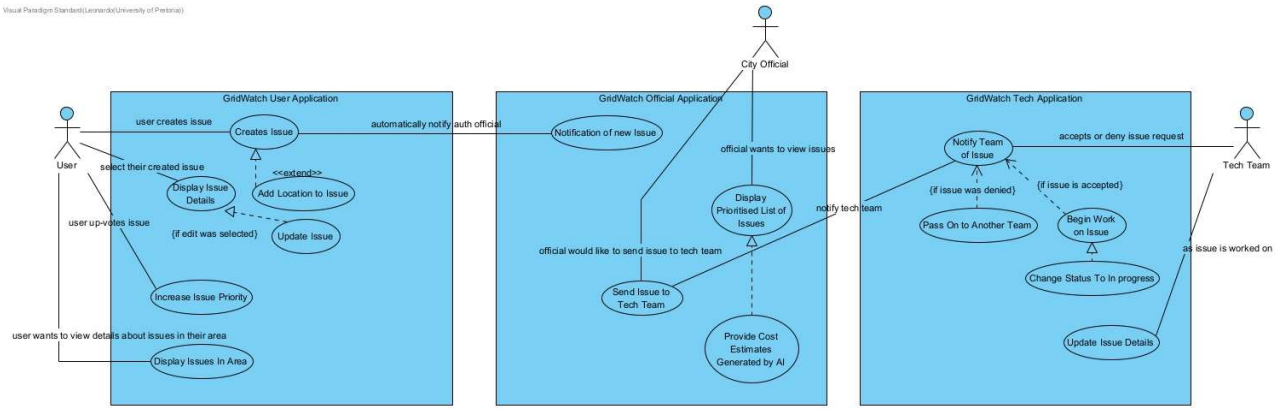
3.1.3.1.1.

4.2.

5.1.

5.1.1.1.2.

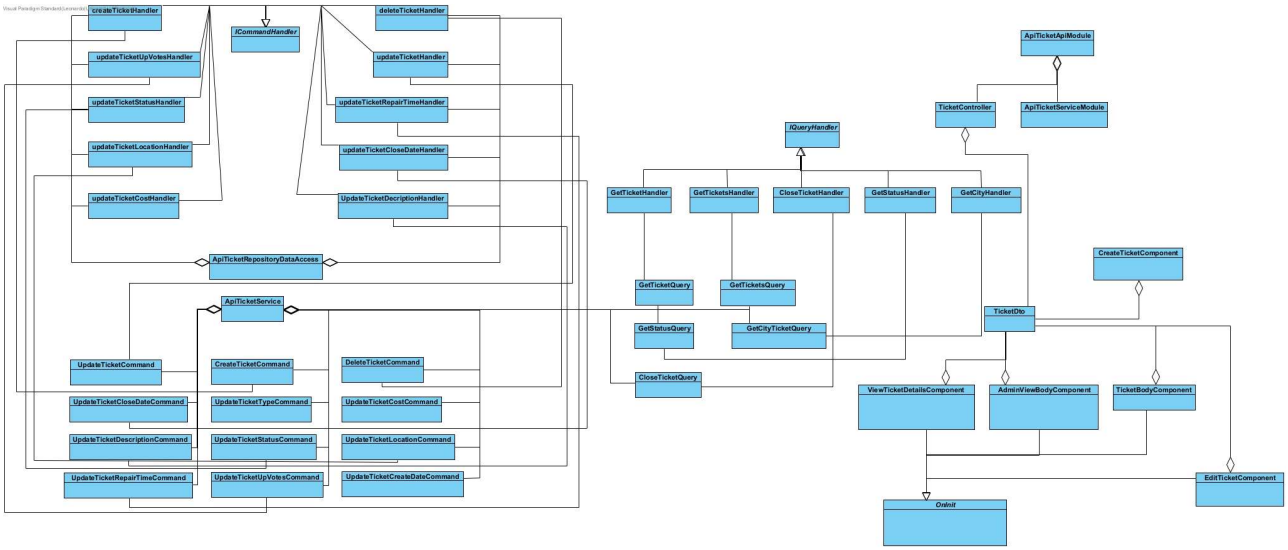
Subsystems



Service Contracts

[Our Contracts]

Class Diagram



Trace-ability Matrix

Requirement	Priority	Reigstration System	Ticket Creation System	Ticket Viewing System	Ticket Update System	City Official System	Technician Team System
1.1	2	x					
1.2	1		x				
1.3	1				x		
2.1	1		x				
2.5	2		x				
3.1	1					x	
3.1.1	3					x	
3.1.2	4					x	
3.1.3	2			x			
4	3					x	
5.1.1	3						x
5.1.2	4						x
6	5						x

User Stories

Epic User Story Template

Epic	User stories		Acceptance criteria
Epic 1: Public	User story 1	**As a** public user, **I want to** create an issue **so that I can** keep up to date with its changes.	**Given that** I create an issue in the application, **when** I have created the issue, **then** my issue should be displayed to me and viewable to others.
	User story 2	**As a** public user, **I want to** pin a location to a task **so that I can** more accurately represent my issues location.	**Given that** I pin a location to my issue, **when** the issue is being created, **then** my issue should be visible to others in the same area.
	User story 3	**As a** public user, **I want to** view other issues in my area **so that I can** view any issues in my surroundings.	**Given that** I load my surrounding areas map, **when** I view the area around me, **then** any issues around me should be represented.
	User Story 4	**As a** public user, **I want to** upvote a selected issue **so that I can** give this issue a higher priority.	**Given that** I upvote an issue, **when** I select an issue I wish to upvote, **then** this issue should be presented before other issues.
	User Story 5	**As a** public user, **I want to** select an issue **so that I can** view more details about the issue.	**Given that** I select an issue, **when** given the option to, **then** the the details of the issue should be displayed.
Epic 2: City Officials	User story 1	**As a** city official, **I want to** view public created issues **so that I can** select issues to dispatch to technician teams.	**Given that** there are issues to view, **when** the city official view is loaded, **then** issues should be displayed to the city official.
	User story 2	**As a** city official, **I want to** be able to sort issues by ticket filters **so that I can** more easily prioritise some tickets.	**Given that** there are issues with priorities, **when** displaying tickets, **then** I should be able to filter these options.
	User story 3	**As a** city official, **I want to** receive notifications of new tickets **so that I can** be aware of possible new higher priority issues.	**Given that** a new ticket is created, **when** the issue is sent, **then** there should be a notification to the city official alerting them of the new tickets.
Epic 3: Technician Team	User story 1	**As a** technician user, **I want to** accept or reject requests **so that I can** provide updates on work or pass an issue on to another tech team.	**Given that** an issue is dispatched by a city official, **when** the issue is received, **then** I should be able to accept or reject the issue as a task.

Architecture

In this section we will discuss the architectures that our system will use.

Design Strategy

Our architectural design strategy choice was to use the decompositional strategy. This strategy means that we are able to generate our approach to choosing architectural patterns by reducing our use cases into sub-systems and identifying suitable architectural patterns to suit the needs of the specific sub-system.

Styles

There are two primary choices of styles that we would like to represent in our system:

1. Service-Oriented Architectures. This style was chosen as the system will need to communicate with an API and to a shared database. Each of these will need to act as a service consumer and sometimes as a service provider. The system will also need to integrate some AI technologies which may not be written in the same language as front-end services or the system API.

2. Layered Architectures. This style lends itself directly to scalability. We intend to use this style to facilitate many users.

Quality Requirements

1. Scalability
   1.1. The system must be able to service requests from up to 100 000 users within 10 seconds.
   1.2. The system must have a serviceable (meaning providing an application which meets core requirements) to the following Operating Systems including Windows, MacOs, Android and IOS.
   1.3. The system should allow for a database to be able to hold up to 100 000 entries.
2. Reliability
   2.1. The system must not, in the case of an unexpected event, lose user information.
   2.2. The system must not, in the case of an unexpected event, lost ticket information.
   2.3. The system should be usable again within 30 minutes of an unexpected event.
   2.4. There should not be a time where the system is unavailable except for an unexpected event.
3. Maintainability
   3.1. Each application should run independent of one another meaning that if one application were to be disabled or not run that other applications should be usable.
4. Security
   4.1. Every password stored should be hashed.
   4.2. Every password should have salt added to it and this salt should be hashed and stored.
   4.3. Every input that would send a query through to a database should be sanitized.
   4.4. A password hash should have an efficiency of 80% over 10 000 entry attempts.
5. Usability
   5.1. Every page which is presented to a user will have a back button.
   5.2. No page presented to a user will contain any dead links.
   5.3. No page should have overlapping content at any screen size.
   5.4. Within an application every page should be reachable using button navigation.
   5.5. On any page text should not be obscured by colours of backgrounds.
   5.6. No button which produces a pop-up will not contain a back button to navigate to a previous page.

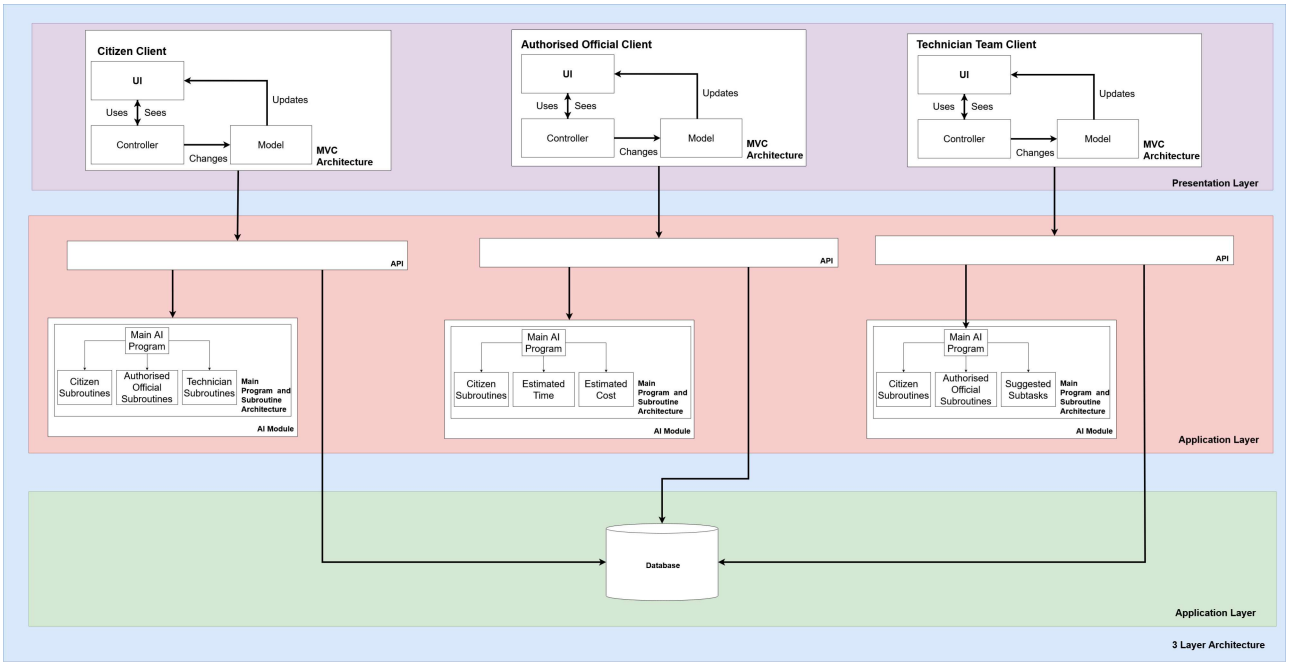
6. Modularity

   6.1. A new application with no API end-points and a hello world entry page should be able to be generated within an hour.

   6.2. Removing a relation from a schema should not effect CRUD operations performed on another relation where the deleted relation was not used to obtain an identifier in the other relation.

Design & Pattern

Below is a visual representation of our entire systems architecture:



A brief re-explanation of the system should be given before continuing. Above we see 3 separate clients being represented which all communicate with the same centralised database. These clients are each 2 layered architectures and each client performs its own operations. Each client, while having a similar structure, each are present to serve a different user of the system.

From here we will present an explanation for each architecture.

1. The use of a layered architecture allows us to separate our front-end sub-systems from our application layers sub-systems. This provides scalability as more layers can be introduced and flexibility as changes need only be made in their specific layer.

2. For our presentation layer we are using an MVC architecture to manage our front-end systems and provide an up to date view for our users. Using an MVC architecture also allows for modular updates and contributes towards our quick loading client.
3. The system will require some AI functionality at each client and to service this we are using a main program and subroutine architecture. This architecture enables us to have one main AI program that can easily be added to and modified.

Constraints

1. Clients developed should be cross-platform. Mobile applications should service both IOS and Android while web clients should service Windows and MacOS
2. The architecture that is chosen should be easily scalable.
3. The system should be able to be hosted using cloud services.

Technology Choices

Here we wish to discuss the different technologies that we will be using and how these technologies were chosen in comparison to some of their counterparts. The intended architecture that we wish to use and its impact will be highlighted in our decision. We have split the technologies we wish to use to different layers

Front End

The choices which were considered for our front-end system were:

1. AngularJS
    - Angular is an open source front-end framework renowned for its efficiency.
    - Pros - Reusable components, two-way data binding, allows for dependency injection and creates responsive websites.
    - Cons - Limited SEO options.
    - How does the technology fit within our architecture? Angular integrates incredibly well with layered systems as its self contained nature can be described in the application layer. React lends is also easily modifiable to service our SOA style.

2. ReactJS
    - React is an open source UI creator by means of using UI components.
    - Pros - Reusable components, create responsive websites.
    - Cons - Poor Documentation and limited SEO options.
    - How does the technology fit within our architecture? ReactJS allows for similar scalability to Angular thus integrating well with our design style.
3. BackBone.js
    - Backbone is a JS library providing key-value binding and custom events.
    - Pros - Lightweight, custom events and more easily communicate with API.
    - Cons - Slow and high debugging difficulty.
    - How does the technology fit within our architecture?

Our final choice was to use AngularJS as this fits with both of our Architectural Styles and will scale effectively while also fitting into a layer of our layered architectural pattern.

Back-end

The choices which were considered for our back-end system were:

1. NestJS
    - This is a NodeJS framework which facilitates the building of server side applications built towards a specific standard or with a specific structure in mind.
    - Pros - Easily test code, easily customisable API endpoints and is self-contained.
    - Cons - Complexity and potential difficulty debugging.
    - How does the technology fit within our architecture? NestJS meets modularity as a quality requirement concern and is very scalable with how easy it is to generate and interact with code.
2. NodeJS
    - This is a back-end runtime environment built on JS to serve application.
    - Pros - Asynchronous calls, NPM package manager and large support.
    - Cons - Single Processor optimised, large number of nested callbacks and difficulty with relational DB.
    - How does the technology fit within our architecture? NodeJS, being a foundation for NestJS, offers similar benefits but is not as scalable and can affect usability when longer operations are performed.

3. ExpressJS
    - This is a NodeJS framework which facilitates the building of server side applications in a customisable and efficient manner.
    - Pros - Well documented, very flexible and widely supported.
    - Cons - Must be strictly maintained to avoid structural issues.
    - How does the technology fit within our architecture? ExpressJS is quite similar to NestJS in that both offer a solution to NodeJS fallbacks however ExpressJS allows a user more potential complexity with higher scale and integrating more features as any pattern can be used to extend your system which can affect efficiency of the system.

The choice here was primarily between NestJS and ExpressJS as the functionality which they provide in addition to NodeJs is essential to our system. Finally our decided technology was to use NestJS as it relates to our styles, strategies and patterns well while also integrating well with AngularJS.

Database

1. PostgreSQL
    - This is a Relational DB Management System (RDBMS).
    - Pros - Highly expandable, SQL standard compliant and has complex data types.
    - Cons - Potentially low entry reading speeds.
    - How does the technology fit within our architecture? PostgreSQL offers largely scalable and efficient database creation and access while being reliable.
2. MongoDB
    - Document-Oriented database program. Structured documents with optional relations are used to generate database.
    - Pros - Wide range of supported languages, high speed and simple to use.
    - Cons - Joining tables for queries, limited size and high memory usage.
    - How does the technology fit within our architecture? MongoDB prioritises efficiency and usability.
3. MySQL
    - This is a Relational DB Management System (RDBMS).
    - Pros - Multi-platform and portable.
    - Cons - Large databases are inefficient, prone to data corruption.

- How does the technology fit within our architecture? MySQL offers usability and simplistic querying.

Finally our decision was to use PostgreSQL as reliability is essential to our system and any efficiency trade off that would be gained by using MongoDB is outweighed by the scalability of PostGreSQL.