UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science

Functional and Architectural Requirements

Client: Gavin Potgieter

Team: CodeBlox

Tshepo Malesela (Bsc: Computer Science)

Lorenzo Spazzoli (Bsc: Computer Science)

Bilal Muhammad (BIS: Multimedia)

Dirk de Klerk (BIS: Multimedia)

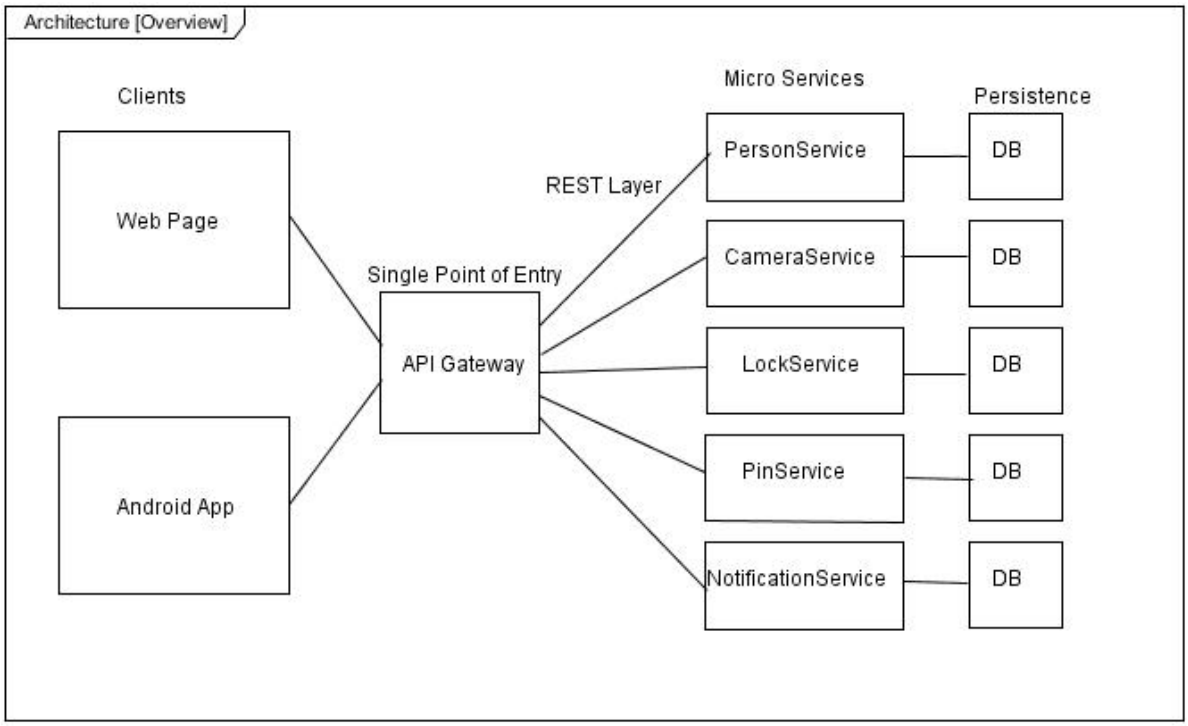
July 29, 2016

Contents

1 Architectural Requirements/Design Specification

1.1 Software architecture overview

The diagram below provides a high-level overview of the software architecture as a whole. For this particular project we are making use of a Micro Services Architecture, therefore the diagram shows how the respective components of the system will be integrated.



1.1.1 The Clients

The component with which the client will be interacting. This will represent the interface with which the client will be interacting, be it an android or web interface.

1.1.2 API Gateway

This component will serve as a single entry point to all the micro services provided by the architecture. The API gateway will be responsible to expose a different API for each of the respective clients.

1.1.3 RESTful layer

Each of the respective microservices will be wrapped in a restful layer, which is then accessed and controlled by the API Gateway.

1.1.4 Micro Services

The micro services are all of the independent functional components that make up the application as a whole. Each of these services aim to complete/address a single task. i.e. each of the microservices will only have a single responsibility.

1.1.5 Persistence

This component is where the microservices will end up storing its information. This particular architecture will make use of a NoSQL database. Each of the microservices will be responsible for maintaining its own persistence as to keep the services decoupled from each other as much as possible.

1.2 Access Channel Requirements

- An android application that allows the owner of the property to be notified when someone is outside the premises and to have a full video stream while the person is on the residence.
- The android application should also allow the owner to remotely provide or refuse access to the premises or dropOff device.
- The application should also allow the owner to generate/provide the person with an access pin, should the camera feed and remote access system not be available at the time.
- Ideally the owner should be able to communicate with the individual/delivery person via voice communication in unison with the camera feed.

1.3 Architectural Scope

- Video stream access of the premises from anywhere at any time i.e. constant camera footage of the premises. This will be done using SIP (Session Initiation Protocol).
- Voice will also be transmitted using SIP, when the delivery person interacts with the intercom system.
- An internal server that will host all of the devices within the household, and that will allow additional devices to be appended as needed in the future.
- An external server that is responsible for establishing communications with the internal server and user application. As well as the responsibility of authenticating and authorizing users.

1.4 Quality Requirements

1.4.1 Performance

- The system should respond to user requests within a reasonable amount of time.
- The live stream delay from the camera should be kept to minimum (ideally no more than a 1 second delay).
- The same delay times should apply for voice streaming should a intercom system be employed.

1.4.2 Scalability

The systems ability to adapt to change in the number of users and should allow:

- the system to handle an increased amount of traffic.
- the amount of resources should be able to be easily increased.

1.4.3 Security

- First and foremost, the system needs to be able to authenticate a user, so that only he/she may access the various services provided.
- Secondly the system needs to be able to authorize the user so that he/she may only use the services for which they are registered.

1.4.4 Privacy

Relates to sensitive user information that they would not like exposed and should not:

- reveal any information about a user that is considered sensitive to any improper parties.

1.4.5 Reliability

The system should provide a reasonable level of availability and reliability.

- The system should provide the user with a fail-safe mechanism to still have access to the basic functionality of the system.
- Allow deployment of services without affecting the neighboring services.

1.4.6 Flexibility

- One should be able to add additional access channels and devices to the system, which is especially relevant to this particular project.
- Add additional functionality to the system with ease.

1.4.7 Maintainability

The system should in future, support maintainability. This will include:

- a system that is easily understood by future developers.
- technologies used within the system should have a reasonable lifetime, as to not force a migration to new technologies in future.
- adding upgrades to the system with ease and minimum downtime.

1.4.8 Monitorability

Relates to how the system is functioning and should allow:

- operators of the system to monitor its operation from a remote location.

1.4.9 Usability

Relates to how users will be interacting with the system and should allow:

- the system to be intuitive and efficient to use with the need to provide additional training.
- the average android or web user should find the system easy to use.

1.4.10 Cost

- The cost of the system should be kept to a minimum and made as affordable as possible.

1.4.11 Testability

- The individual components of the system needs to be testable through the use of mock objects and automated unit tests.
- Automated integration tests should be used to test the system as a whole.
- In particular the tests need to verify that the preconditions are met, and that post conditions hold true once a service has been provided.

1.5 Architectural Constraints

The client has some architectural specifications in place that will need to be adhered to in order to successfully complete this project.

1.5.1 Use of Inexpensive Devices

The devices used to construct the physical system needs to be as inexpensive as possible, thus providing a larger audience with access to it. Devices such as the following will be included:

- Raspberry Pi 3
- IP Camera/Pi Camera
- Internet Enabled Switches
- WiFi Routers
- Android Smartphone (Users are assumed to posses one already).

1.5.2 Technology Stack

For monetary reasons, the client has requested that the entire system be developed using open source technologies. CodeBlox has therefore decided to settle on a JavaScript Ecosystem, since it addresses performance and scalability requirements intrinsically through the use of the MEAN stack. The MEAN stack refers to MongoDB, Expressjs, Angularjs, Nodejs. A Preliminary stack will include the following technologies:

- **JavaScript** The primary language to be used throughout the development of the system.
- **Nodejs and Express.js** as the cross-platform runtime environment that supports the creation of new modules using JavaScript. Express will also have the responsibility of making the services RESTful.
- **Android Studio** to the build the mobile application that will be used.
- **npm Scripts** will be used as the primary build tool and will be in charge of keeping track of the life cycle.
- **Mocha and Chai** will be used for unit testing and assertion
- **JAX-RS** for implementing RESTful webservices.
- **Linux** as the operating system
- **MongoDB** as the primary persistence technology.
- **Mongoose** as an ODM to provide additional functionality when working with MongoDB.

- **AngularJS** as the primary framework for developing a web application.
- **Heroku** will be used to host the application online.

1.6 Architectural Design

This section specifies the architectural responsibilities of various architectural components.

1.6.1 Architectural Responsibilities

- The responsibility of providing an execution environment for business processes is assigned to an application server. i.e. **Nodejs/Express** itself.
- The responsibility of persisting domain objects is assigned to the the database i.e. **MongoDB**
- The responsibility of providing access to the database and converting between POJO's and JSON is left to the ODM i.e. **Mongoose**
- The responsibility of providing access channels to users is assigned to the web services framework i.e. **Express.js**
- The responsibility of providing users access via mobile devices is assigned to the mobile application framework i.e. **Android Studio**

1.6.2 Infrastructure

For this particular project, the **Microservices architectural pattern** will be used as an infrastructure. Although no clear definition exists for the pattern it can be described in terms of it attributes:

- **Software can be broken down into multiple component services.** Thus each service can be deployed, tweaked, and redeployed independently.
- **Services are organised around business capabilities and priorities.** Thus each team/member can develop and maintain services independently.
- **Functions like a classical UNIX system.** Thus it receives requests, processes them and responds to them accordingly.
- **Favours decentralised governance over centralised governance.** Allows developers to produce useful tools that can be used by others to solve the same problem; use existing tools to solve own problems; Each service manages its own unique database.
- **Designed to cope with failure.** Neighbouring services should continue to function independently of the failed service.
- **Focusses on evolutionary design.** This is extremely useful when you can't anticipate the types of devices that may need to access your application in future.

1.7 Application Server/Micro Services

The Micro Services component of the architecture is where all the business logic of the system will hosted. Each of the services within the system will address as specialised need/function. The Micro Services themselves will be implemented via Nodejs/Expressjs.

1.7.1 Quality Requirements

- **Performance**
  - The system should respond to user requests within a reasonable amount of time.
- **Scalability**
  - The server should be able to cope with a reasonable increase in the amount of users using the system for the amount of resources available.
- **Maintainability**
  - The system should be easily maintainable by future developers with as little downtime as possible.
- **Flexibility**
  - New functionality/services should be able to be deployed independently of other existing services.
- **Reliability**
  - Requires that service requests are not partially executed. Thus the requested service will only be executed once postconditions are fulfilled. If a service could not be provided, a reason must be given.
- **Security**
  - The application server must support role-based authorisation. Thus only a user with a certain level of authorisation may execute certain processes. No direct access should be provided to the database.
- **Testability**
  - The application server should support out-of-container testing as well as unit testing.

1.7.2 Tactics as provided by Nodejs and Micro Services Architecture

Node.js/Expressjs and the Micro Services Architecture itself, address the quality requirements mentioned above intrinsically.

- **Performance Tactics**
  - The Node.js Runtime Environment interprets JavaScript using Googles V8 JavaScript Engine. The engine compiles JavaScript to native machine code before executing it, instead of using more traditional means such as interpreting the code to provide increased performance.
- **Scalability Tactics**
  - Node.js has an event-driven architecture capable of asynchronous I/O. This allows the creation of highly scalable servers without using threading, by using events-driven programming that uses callbacks to signal the completion of a task. This architecture aims to optimize scalability in web applications containing many I/O operations, as well as real-time Web applications.
- **Maintainability Tactics**
  - Since the Micro Services Architecture focusses on dividing the application into functional services, it inherently improves maintainability by allowing modules to be modifies and redeployed independently.
  - Nodejs uses a modular approach to programming that supports this architectural pattern.
- **Flexibility Tactics**
  - The independence of each service as well as micro services' tendency to favour decentralised governance, mean that new functionality can be added without affecting the existing services.
  - The notion of evolutionary design is another factor that improves the systems ability to be flexible.
- **Reliability Tactics**
  - Each service of the system should be executed in full once each of the preconditions have been met. If such a service can not be provide, a suitable reason must be given/exception thrown.
- **Security Tactics**
  - The provide the system with sufficient security. His/Her user credential will be exchanged for that of a JSON Web Token (JWT), which will then be saved locally. This token is then used for all future transactions and will be sent with every request.

- **Testability Tactics**
  - The Mocha and Chai npm modules provide a comprehensive testing framework and assertion library. Used with npm Scripts, a fully automated unit and integration test suite can be developed that can be executed as part of the build life cycle of the system.

1.8 Persistence

For this particular project we will be utilising a non-relational database, namely **MongoDB**. MongoDB is a free and open-source cross-platform, document-oriented database. Classified as a NoSQL database, MongoDB avoids the traditional table-based relational database structure in favour of JSON-like documents with dynamic schemas.

1.8.1 Quality Requirements

- **Performance**
  - The database should return the necessary information/JSON objects within a reasonable amount of time.
- **Scalability**
  - The database should support horizontal scaling across an array of hardware to cope with an increased number of users.
- **Security and Privacy**
  - The information stored within the database should be secure at all times.
  - Sensitive user information should be kept private from unauthorised parties.
- **Accessibility**
  - The required information should be easily accessible from the database.
- **Availability**
  - The data should always be available for retrieval and use.

1.8.2 Tactics as provided by MongoDB

MongoDB addresses many of the quality requirements intrinsically through the way it has been designed and implemented.

- **Performance and Scalability Tactics**
  - MongoDB is a NoSQL database, particularly a document-oriented database. These databases assume that documents encapsulate and encode data in some standard format, in this case binary forms of JSON. NoSQL databases include some forms of sharding/partitioning which enables them to be scaled out on hardware (servers) or the cloud. This provides them with massive scalability potential, as well as higher throughput and lower latency, hence improved performance.
- **Security and Privacy Tactics**
  - To be a completed.

- **Accessibility Tactics**
  - MongoDB avoids the traditional table-based relational database structure and uses JSON-like documents with dynamic schemas instead. This makes the integration of data in certain applications easier and faster. Thus it provides developers with an easier way of storing and accessing data since fields can vary dynamically. Mongoose ODM is used to store objects within the MongoDB database.
- **Availability Tactics**
  - MongoDB provides a user with Automatic Failover. This is the process of switching operations to a redundant or standby server or hardware component, if the primary server or component failed for some reason. This will ensure that the data required by the system will always be available.

1.9 Mongoose

Mongoose is a Node.js library that provides MongoDB with Object Document Mapping, which is the ORM version of non-relational document-oriented databases. Mongoose translates data in the database to JavaScript objects for use in the application and vice versa.

Additional Quality requirements addressed by Mongoose:

- **Reduction in code bulk:** ODM provides numerous services thereby allowing the developer to focus on the creation of business logic as opposed to writing repetitive database queries.
- **Changes to object model are made in one place.** Once an update has been made to object definitions, the ODM will automatically use the updated model for future queries.
- **Caching.** Entities are cached in memory thereby reducing the load on the database.

1.10 RESTful Webservices

REST is an architectural style for networked hypermedia applications, it is primarily used to build Web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service. REST is not dependent on any protocol, but almost every RESTful service used HTTP as its underlying protocol.

Every system uses resources and these resources can take the form of any media. The purpose of a service is to provide a window to its clients so that they can access these resources i.e. expose these resources.

RESTful sevices should have the following properties:

- Representations
- Messages
- URI's
- Uniform interface
- Stateless
- Links between resources
- Caching

RESTful sevices will be implemented via the Node Express.js library.

1.10.1 API Gateway

The API Gateway enables the system to expose a different API for each client. This will enable control over which services are available to which user in the context of which services have been paid for. The API Gateway can also be used to implement security in terms of verifying that a client is authorized to perform the request.

1.10.2 Client - Web

Still under revision - To be implemented using AngularJS.

1.10.3 Client - Android App

Still under revision - To be implemented using Android Studio.