Smart NFC Card Application

Architectural Design and Justification

Vast Expanse (Group 7)

COS 301 - 2019

Wian du Plooy	*u17237263*
Duncan Vodden	*u17037400*
Tjaart Booyens	*u17021775*
Savvas Panagiotou	*u17215286*
Jared O'Reilly	*u17051429*

Table of Contents

Introduction

Vision

Link is a mobile phone application used to simplify and enhance business networking and networking/interaction between companies. Contact information can be shared, and potential clients can be hosted at the company offices, all with the simple tap of a phone. The strong new wave of NFC technologies emerging is the driving force behind Link - enabling easy and effective new ways to communicate and share information, wirelessly.

We hope Link will become the new industry standard for networking between companies. Using this mobile application, companies (and, more importantly, the people within companies) will be able to connect with people in other companies with ease and scale never seen before in business networking. Link is here to link companies together.

Objectives

Link needs to fulfil 2 key objectives to be used effectively as a tool with which to network with employees of other companies:

- Enable easier sharing of company details and employee contact details, as well as make it easier to find the company and where to go to visit the company offices.
- Enable easier hosting of visiting clients at the company offices by allowing simple setup of physical access, guest WiFi access and means to spend money in-house for a visiting client and share it with them.

Link aims to mitigate the time and effort needed for various traditional networking processes:

- The usual hassle of keeping business cards in wallets or purses
- The manual communication of details over WhatsApp or email or related technologies
- The time-consuming organization needed for a client to visit the office premises

Business Need

Networking between employees in companies is needed to ensure these different companies can get in touch with each other and set up negotiations with each other on how their companies can work together in their common interest. Therefore, effective and easy networking will enhance the ease of initial company-company linking, as well as subsequent company-company interaction and negotiation.

Link fulfils these business needs. With its capability to share vital company and employee information between employees, as well as its capability to setup hosting for visiting clients, Link eases and enhances the business networking process, cementing itself as a valuable tool to allow better company-company interaction.

Scope

Link will be a mobile application, paired with a backend-access system, which will be accessed ideally through a web interface. The mobile application and backend-access system will allow controlled altering and viewing of the data stored on a backend database, with this database being an efficiently and logically designed relational database scheme using an appropriate relational database management system (RDBMS) such as PostgreSQL. To provide the access to this database, it will be securely wrapped and accessible through a web API, implemented using an appropriate server-side language such as NodeJS. The mobile application will be developed using a web framework specifically designed for cross-platform mobile application development, such as the Angular-Ionic-Cordova framework.

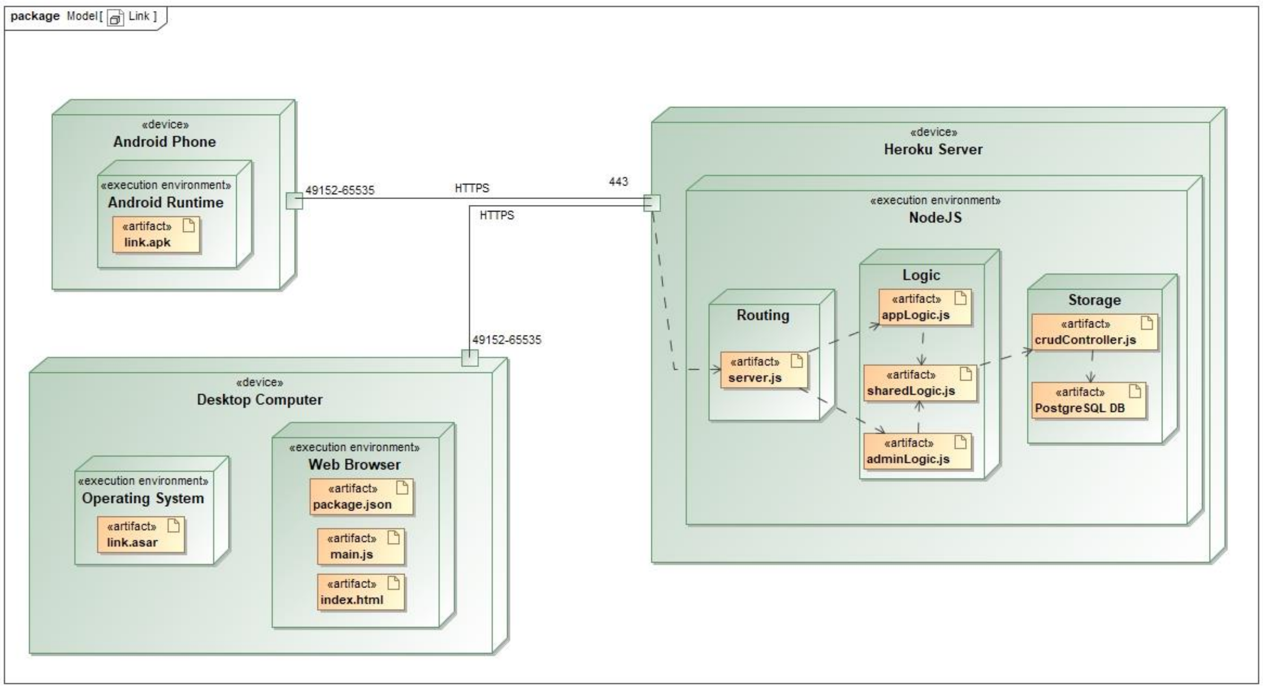
Link will allow employees of a company to share their virtual business card as well as their company's location with potential clients who also have the Link app, with some simple setup beforehand, followed by a simple tap of phones to effortlessly share vital contact information. Link will also simplify the process of hosting a potential client at a company's offices, by enabling temporary physical access into the premises, temporary guest WiFi access and temporary cards with which to make in-house purchases to be easily setup using the app and then shared with visiting clients, with a satisfying tap between phones.

All setup of business card and location sharing, temporary physical access into the premises, temporary guest WiFi access and temporary cards for smart payments will be able to be done in the mobile app by employees of the company, and will be able to be shared using the mobile app from employees to potential clients (who may themselves be employees of other companies). Only Link administrators will be able to use the backend-access system, to insert new companies and employees and other details required for when a new company starts using Link, as well as to draw reports on transaction logs for the smart payment cards.

Definitions

Acronym	Definition
BAS	Backend-Access System
VBC	Virtual Business Card: The digital version of a business card that is used by our system instead of a physical business card.
TPA	Temporary Physical Access: A virtual version of an NFC enabled access card that is used by our system to grant access to a client.
TGW	Temporary Guest WiFi: WiFi access given to a guest for a limited period of time.
RDBMS	Relational Database Management System
API	Application Programming Interface
NFC	Near Field Communication
CRUD	Create, Read, Update and Delete

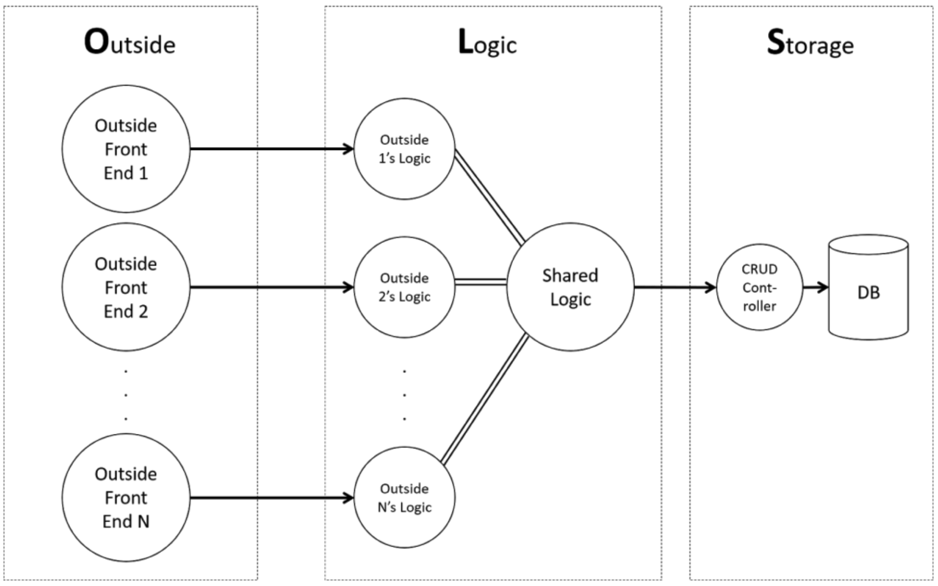
Deployment Diagram



Architectural Designs and Requirements

The OLS Architecture Design Pattern

For the Link project, we have created our own architectural design pattern, which we believe is not only useful for the Link project but for many currently existing software applications and for future software applications. Our custom architectural design pattern is called **OLS**, which stands for **O**utside, **L**ogic, **S**torage. **OLS** consists of 3 components (O, L and S) and each of these components uses an architectural design pattern within themselves. The below diagram illustrates the core components of **OLS**:



The 3 distinct components of **OLS** (O, L and S) can be described in the following way:

Outside

Represents:
The Outside component represents the various front-end subsystems, designed for the users of the project. Each outside front-end is designed for a specific user type and may be hosted on a different type of device.
The choice of front-end framework/languages/model is dependent on the project and which framework would suit the project well. The only thing required is the ability to make requests to an endpoint, e.g. an HTTP request.

Communication:
These outside front-ends will only know of their associated Logic sub-component (located in the greater Logic component), which means they will only need to communicate with this one sub-component and use the API functions defined in that sub-component, without worrying about the rest of the architecture.
This one location of communication greatly simplifies requests sent out by this outside front-end, as it only must know the API functions of one sub-component and the location of that sub-component.

Architectural Design Pattern:
The architectural design pattern used here can vary, allowing a software architect to make a good problem-dependent choice.
For example, a variant of the **MVC architectural design pattern** could be used here, as it enables Views to be created with appropriate Controllers (as this is the 'front-end' component, this is important) and the Model could be the connection to the Logic sub-component, fetching relevant data in storage and making appropriate changes to the data in storage.

Logic

Represents:
The Logic component represents the API's used by the various front-end subsystems, with each front-end interface having an associated sub-component (API) in the greater Logic component (which all their requests sent are sent to).
These sub-components utilize a single sub-component, called Shared Logic, which stores a collection of useful functions to be used (e.g. regex validation functions, password hashing functions) as well as a collection of functions which deal with, strip and extract requests coming in and also formulate the responses to these requests. Shared Logic also provides the connection to the Storage component.
This grouping of functions into Shared Logic allows each sub-component to only have to implement the required functionality it needs to present to its associated outside front-end and not all the necessary pre and post-processing tasks, simplifying the coding of these sub-components. Shared Logic also allows certain standards all sub-components should follow to be defined in one place, in the Shared Logic sub-component, simplifying standards updates and increasing maintainability and updating.

Communication:
These sub-components in the Logic component only need to communicate with their respective outside front-end and with the Storage component, and both of these communications is done through the Shared Logic sub-component, meaning these sub-components can provide functionality regardless of the location of the outside front-end and the Storage component - they only need to know where Shared Logic is.
This one location of communication greatly simplifies the responses that the sub-components need to create, as well as how they would use the Storage component, as it is done through one channel.

Architectural Design Pattern:
The architectural design pattern used here is the **N-tier/Layered architectural design pattern**. There are approximately 3 layers used in this Logic component: a *Routing* layer (which is implicit as well as trivial, simply directing an outside request to an endpoint on a sub-component to the appropriate sub-component, so it can deal with the request), the *Sub-Logic* layer (which contains the sub-components that deal with requests routed to them by the *Routing* layer), and the *Shared-Logic* layer (which contains the Shared Logic sub-component, providing simplifying functionality to the sub-components in the *Sub-Logic* layer, as well as receiving the responses from the *Sub-Logic* and sending them back to the relevant outside front-end, and finally also receives requests to fetch/update data in the Storage component from the *Sub-Logic* layer and forwards it to the Storage component).

Storage

Represents:
The Storage component represents the persistent storage used by the project, in the form of a database and an API wrapper (called CRUD Controller) which facilitates controlled access into the database and controlled modification of the database.
The database can use any DBMS appropriate to the application, and CRUD Controller must provide functions to create, view, update and delete the data in the database, with parameters of the logical data that needs to be used. This way, only the CRUD Controller needs to know the exact structure of the database and which language to use to query the data, decoupling the rest of the project from the DBMS used. All communication into the database is therefore received by the CRUD Controller and a response returned by CRUD Controller.

Communication:
The CRUD Controller provides an interface to the sub-components of the Logic layer (through the Shared Logic sub-component) to access the database, and directly accesses the data in the database using the correct query language. It does not need to know where the Logic layer and its sub-components are, only where the database is located.
Having one entry point to the Storage component (the CRUD Controller) simplifies the access into the database and allows the DBMS and database scheme to be changed with only CRUD Controller having to update its code. This logical CRUD of the data in the database allows the rest of the project to communicate logically with the Storage component, simplifying the querying of the database, only through one point of communication.

Architectural Design Pattern:
The architectural design pattern used here is the **Persistent Storage architectural design pattern**. The database is where we store our data persistently, and the CRUD Controller is how we access this persistently stored database. This is the standard approach to using a database in this type of application and has been tried and tested with good results. Having more than one database (distributed) or changing the scheme of the database is also possible, all that has to update is the CRUD Controller (which ‘translates’ the logical requests for data into the explicit queries for data and returns the result).

Advantages of using OLS

The core components of OLS were described above, however, this general description hides the reason why we created and chose to use this architectural design pattern.

OLS Integrates Easily

The Link application started off with a simple description - a mobile app and an admin portal, paired to a backend server. However, we soon discovered that our application would have to integrate with many other systems and interfaces, not just an admin interface and the mobile app.
For example, we need to integrate our system with the access control system of the EPI-USE offices and other access control systems of other companies. We also needed to integrate our system with WiFi routers of the companies in their buildings. It would be difficult to understand the inner workings of each system to be integrated into, so instead, for every system to integrate with (i.e. each Outside Front-End), we can define a new sub-component in the Logic component, which that Outside system would use to gain access into our data in a controlled fashion.
This approach means that the outside system to integrate with only requires one additional API, meaning we don't have to change any existing ones and this API will solely serve that outside system. A new integration will therefore add as minimal code as possible.

OLS Prevents Re-Definition and Standardizes

With the use of the Shared Logic, any and all general functions which all the Sub-Logic components need can be defined in one place (in the class inside of Shared Logic). This means they don't have to redefine the methods in their component, and this also means that if we need to change what one of those general functions do, it can be changed in one place and all other components will be able to continue using the functions without any change.
Not only does using OLS prevent re-definition (as described above), it also allows for certain standards for the API's to be specified once and implemented once (in Shared Logic). These standards can include the format of JSON to be sent in the body of a POST request, the variables to use in the classes to access the body of the POST request after extraction and parsing, the formatting of responses to requests, the way we validate websites, cellphones, emails, passwords, etc. The entire system, using OLS, can therefore utilize these standards with no exception, and they only must be defined once.
This standardization avoids different software engineers doing their own formatting and following their own standards, differing from the team's approach and eventually leading to conflict and code mismatch errors. In OLS, standardization and definition are defined once, in one place - as semantically they should be.

OLS Separates the Database

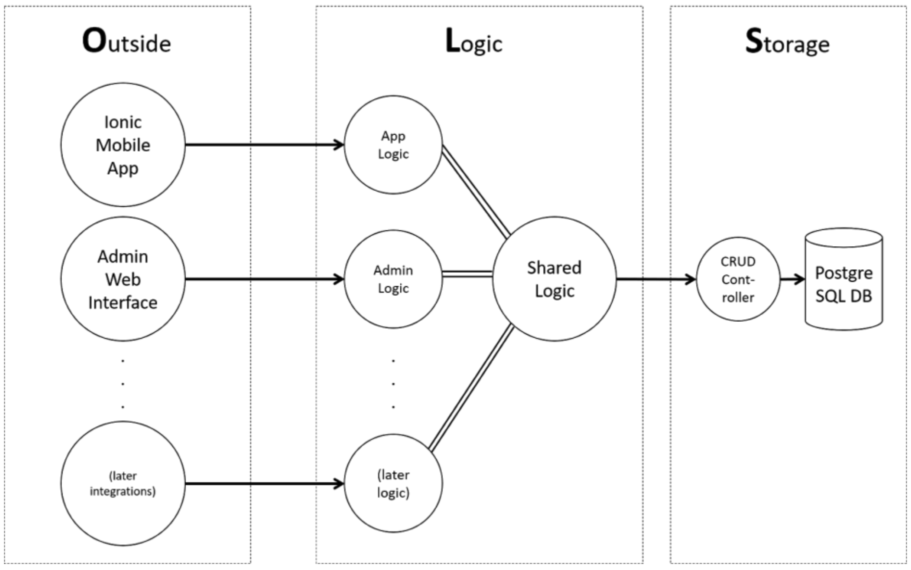
As our data grows bigger and bigger, new types of databases have been developed to improve the throughput of our database systems (like NoSQL). However, many of the applications present today started off with Relational Databases and had to change much of their code in a very non-obvious and non-trivial fashion, resulting in much time and money spent on effectively only upgrading the database system.
OLS, due to the use of CRUD Controller, allows an interface to the database to be used by the other components, with 'logical' queries being executed (for example, not 'SELECT * FROM tblEmployees WHERE….', instead, one would call getEmployee(..)). This means that the structure and scheme and type of the database is completely decoupled from the rest of the system, allowing the database to be completely switched out to a new type/scheme or the structure to change, with the only changes needing to take place in the CRUD Controller.
This means that updating our code to use a new database system is easy - we only need to look in one component and update that, maintaining the same interface. So, when, in the future, new database systems come out that handle our data in new and innovative ways, to update the code of

an OLS-architecture based system would be easier than other architectures and save both time and money.

OLS is in Demand
The usage of mobile applications and admin interfaces, paired with a backend API, is very prevalent in the field of software engineering is high. These types of projects are common and in demand, and these created systems needing to be integrated with other systems is happening more and more in present-day software society, as all our various systems get bigger and talk to each other more and more (e.g. the Internet of Things). OLS provides an easy way to conceptualize, design and implement these types of applications, providing lower coupling and better maintainability than other traditional architectural design patterns.

Our Use of OLS

After the above description of OLS, we can now illustrate how we are currently using the OLS architectural design pattern in our project:



Our Outside component consists of:

- Our Ionic mobile application, to be tested with Android devices initially
- A web interface for Link admins and company admins to access the system
- (possible further systems that integrate with us and our system)

Our Logic component consists of:

- App Logic, the API that the Ionic mobile app will request to
- Admin Logic, the API that the Admin web interface will request to
- Shared Logic, a class that extracts and parses the JSON of a certain standard in HTTP POST request bodies, provides API Key validation and Login functionality, provides some general validation functions (e.g. to determine valid emails), provides functions to format the response to a request in the set standard and provides access, through the CRUD Controller, to the PostgreSQL database.

Our Storage component consists of:

- CRUD Controller, the function API that all other sub-components in the Logic component will use to logically query the database
- A PostgreSQL database and DBMS, enabling us to use the entity-relationship capabilities of a relational database to simplify and optimize our data storage

Constraints

The architectural constraints are separated based on the three major components of the **OLS** architectural design:

Outside (O) Constraints

- Each outside component must communicate with its respective Logical Controller responsible for the server-side logic
- Each outside component must comply with the data transfer requirements set by its Logical Controller, this is set by the producers of the individual controller and is not necessarily standard between Logic Controllers.
- Each outside component must have an end point to communicate to its corresponding Logic Controller which can differ based on the different types of outside components

✓ **Disadvantages:** High coupling between the Outside Components and their respective controllers

✓ **Advantages:** Separation of Concerns

Logic (L) Constraints

- A single point of failure exists in the Shared Logic controller in the Logic component. All logical controllers that make use of this dependency have the potential to crash if the Shared Logic controller fails.
- All access to data in the Storage Component must be sent to a CRUD Controller, no direct database access is permitted.

✓ **Disadvantages:** Multiple logical layers increases integration complexity and can produce increased communication overhead, however this allows for Separation of Concerns.

✓ **Advantages:** Separation of Concerns between the logic of different outside components, decreased coupling within the Logic Component.

Storage (S) Constraints

- **Input Constraints**
  - All communication with the Storage component must enter through the single point of entry known as the CRUD Controller.
  - All input data must comply to the standard data transfer format of the architecture's implementation, which is the JSON format. This standard format can change in other implementations of the architecture.

- **Output Constraints**
  - All responses from the CRUD Controller must comply with the specified standard JSON format.
  - If the implementation makes use of multiple (distributed) CRUD Controllers, then these controllers must have a protocol in place which keeps the database in a stable state. Thus, the output of two identical queries should not be different unless in between these two queries there was a modification to the database.

✓ **Disadvantages:** As it currently stands, there is a single point of failure for data access to/from the persistent storage. This is rectified by the ability for the Storage Component to have multiple distributed controllers with the same functionality which will provide a nearperfect distributed access for the Logic Component. This solution however leads to the need to update multiple instances of the CRUD Controller in multiple places which could be a daunting task. Orchestrated deployment tools such as Kubernetes can be used to solve this auxiliary problem.

- ✓ **Advantages:** These input constraints allow for increased security as the CRUD Controller acts as a proxy between the Logic Layer and the actual database. This proxy ensures that all queries that reach the database follow the specific constraints imposed by the database management system (DBMS). The CRUD Controller is responsible for sanitizing all queries and constraining the number and type of queries that may be performed on the database by the Logical Layer (L) requester.

Technological Decisions

Version Control: Git – Hosted on GitHub

We will be using git and GitHub as it gives us all the required functionality for version control for our project. It allows us to track changes in our code across various versions such as who did what, when and where those files are stored.

Project Management: ZenHub

ZenHub allows for seamless integration with GitHub, allowing us to have our project management tool in the same location as our repository. It has a very easy-to-use interface for creating new features as well as moving features across different stages of its lifecycle.

Continuous Integration: Travis CI

Travis CI is a hosted continuous integration service which is used to build, and test software projects hosted on GitHub. It is easy to use, and we don’t have to think about Travis CI once it is set up. When we commit a change, a build will be triggered and Travis CI will test the integration, if it passes the test, then the integration went fine, if not then Travis will let us know that something went wrong.

Testing: Jasmine, Karma

Jasmine and Karma are open source testing frameworks for JavaScript. They allow for simple integration with our Continuous Integration tool (Travis CI). These testing frameworks allow us to easily execute tests locally during development in order to ensure our code is working and producing the correct outputs.

Front End Framework: Angular 7 and Ionic

We have decided to use Angular and Ionic for our Front-End Framework as it is a cross-platform framework, allowing us to create a single app which will work across various platforms such as PC/Android/iOS (some issues surrounding iOS payments however) without any additional development time for each platform. Using Angular makes it possible to build fast and scalable applications, coupled with Ionic allows for the creation of a powerful and feature-rich application.

Hosting: Heroku

We will be utilizing the free hosting service provided by Heroku as it allows for easy integration with GitHub allowing us to auto deploy our Master Branch (located on GitHub) to Heroku.

Data Storage: Postgres Database

Heroku allows us to utilize a Postgres Database provided the hosting provider. This allows for easy integration with our application, as well as easy setup of the database itself.

Server: NodeJS
We decided on using NodeJS as our back-end server it has been proven to be a robust language allowing for increased speed and performance with comparison to other languages. There are a vast number of tools that we can utilize within NodeJS in order to complete our project. As a plus side our team is experienced and comfortable in using NodeJS as our server language

Mobile Technology: NFC – enabled Devices
For the purpose of our project we require NFC-enabled devices. This will be used for the core functionalities of our project for sharing data and information between mobile devices.

Quality Requirements and Justifications
The requirements in this section provide information about the quality of the application and what the application should be able to achieve.

Q.1 Performance
Q.1.1. Link must be able to handle 200 requests per second.
Q.1.2. Link must be able to respond to an initial request in under 1 second but will depend on internet connection strength and location of the server.

This will ensure that users of Link will not get frustrated and wait in queues to access the application and that they also do not have to wait long for a request to be handled.

Q.2 Reliability & Availability
Link will be hosted on an external server with a contractual agreement with the service provider.

Q.2.1. Availability
Q.2.1.1. Link must be available for 99% throughout its lifetime.
Q.2.1.2. Link must have access to the databases for 99% throughout its lifetime.

This will ensure that all users will be able to use the application as much as possible. It will provide trust between the user and the application as the user will always be able to use it.

Q.2.2. Reliability
Q.2.2.1. Link must behave the same in deployment as it did in testing.

This will ensure that the application behaves in an expected manner. The application must behave the same way in the development stages for every user, as it did when the product owners conducted tests on the application.

Q.3 Extendibility
Q.3.1. Link must be designed using appropriate design patterns to allow for easy extension of functionality.

Allows the application to be extended with more functionality without changing lots of classes or modules. This allows for an easy to maintain and easy to fix, design for the application which is desirable.

Q.4 Usability

Q.4.1. Graphical User Interface

Q.4.1.1. Link must have an easy to navigate user interface to allow all users to understand the application.
Q.4.1.2. Link must be designed in a vertical approach rather than a horizontal design to make navigating through the application easier.

This will ensure that users will find the application easy to use and navigate. It will also be responsive so that it is visually pleasing for the user.

Q.5 Security

Q.5.1. Data Storage

Q.5.1.1. The data of users must be stored in a secure manner and must have controlled access.
Q.5.1.2. All data that conform to the Customer Laws need to be logged for the required amount of time and must be deleted after a certain amount of time.
Q.5.1.3. Audit logs must be stored and must only be accessible to the product owners.
Q.5.1.4. Client passwords must be hashed and salted before storage.

This is to ensure that all personal information that is stored on the database can't be accessed by users of the system. It will also log required information about people as is required by law and will remove these logs after a certain time as is required by law.

Q.5.2. Data Transfer

Q.5.2.1. Data sent over the internet must be encrypted and securely transferred between different locations.
Q.5.2.2. Data sent over **NFC** must be encrypted.
Q.5.2.3. When sending data over **NFC**, a user must be prompted to transfer the data or deny the transfer of data.

This is to ensure that no personal data can be obtained by people in the middle of data communication. This will ensure encrypted data communication to promote privacy. It also allows a form of two-factor authentication in the sense that a user will be prompted to accept data sent over NFC, which adds an additional layer of security.

Q.5.3. Data Access

Q.5.3.1. All data must have clearance levels associated with it, which will give controlled access to data.
Q.5.3.2. All data logs must have controlled access and can only be accessed through an interface (not **API**), only users with desired clearance levels may access the data.
Q.5.3.3. The owner of the product must be able to add privileges or remove privileges from a client.

This will prevent data from being accessed by people without the necessary clearance levels and keep the data private. It will also prevent users to remove any logs and therefore can't hide something by deleting logs.

Q.6 Testability

Q.6.1. All features offered by Link must be testable through unit tests.
Q.6.2. All subsystems of Link must use integration tests to test whether all features it needs from other subsystems are available and if they work.

This will ensure that the application behaves as it should during the testing phase, meaning that you can easily find mistakes in the application.