[arc42] deChat - Decentralized Chat 0

NOTE	This version of the template contains some help and explanations. It is used for familiarization with arc42 and the understanding of the concepts. For documentation of your own system you use better the *plain* version.

1. Introduction and Goals

The development will consist of a WebApp whose objective is to be a decentralized chat, using SOLID technologies. This chat will allow users to exchange messages in different formats (audio, text, video), as well as other functions such as: share screen or video call; It will also allow you to save friends lists, show notifications, among other functions. It will be a web application developed at Angular, and an RDF library will be used to communicate and obtain information on the different PODs

1.1. Requirements Overview

The application will mainly comply with the following:

- The system will be based on a decentralized architecture where data storage is separated from the app
- Users can store their chat data in their own pods
- The app will allow a user to share pictures, videos or other kinds of files with other friends through the chat
- A user can get notifications when some friend wants to chat with him
- Users can have groups of friends with whom they may want to chat
- It will be possible to have group chats where all members receive the messages
- Users can have live calls and videos
- Users can share their screens in the chat

1.2. Quality Goals

The main Stakeholders of this application are Inrupt and users, I will list the most important quality goals for them:

- Usability : Ease of use by non-technical people.
- Responsiveness : The application must send messages in real time.
- Privacy: The application must keep the data private (this is what solid is based on, each one owns their data).
- Security: The application must be secure, users must know who they are talking to and who has access to the chat.

1.3. Stakeholders

Stakeholders of the system:

- Developers
- Professor
- Users
- Inrupt

Role/Name	Contact	Expectations
Developer	Laura Sanchez	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Developer	Enrique Fernandez	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Developer	Jose Luis Bugallo	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Developer	Victor Manuel Chaves	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Developer	Carlos Gomez	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Developer	Daniel Fernandez	Have to work with the architecture or with code, documentation of the architecture,have to come up with decisions about the system or its development
Professor	Jose Emilio Labra	Evaluate and guide the project
Users		Users can chat,they can have live calls and videos and they can share their screens in the chat
Inrupt	Inrupt	They has been proposed as a challenge

= 2. Architecture Constraints Doc Writer <UO244741@uniovi.es> v1.0, 2019-08-02

== 2.1 Technical Constraints

Code	Restricction	Motive
TC001	Program language: JavaScrip or TypeScript	Internet has a lot of code examples
TC002	Frameworks: Angular	The software developers which develop Solid applications are familiar with this frameworks
TC003	Online independence: no central control of data	Linked-data are completely decentralized and fully under users' control rather than controlled by other entities.
TC004	Live calls	Users can have live calls.
TC005	Video calls	Users can have video calls.

== 2.2 Organizational Constraints

Code	Restricction	Motive
OC001	About Version Control: Git, Tool: Github	Keep a version of the project after making changes

== 2.3 Political Contraints

Code	Restricction	Motive
PC001	Security	Data must not be in touch by external devices
PC002	Privacy	Only own users can manage their own data
PC003	Originality	The app must be different from other apps and be recognize
PC004	Usability	The documentation of the app must be easy to understand by users
PC004	Friendliness	The app must have inviting interfaz

```
== System Scope and Context
```

The project that we are going to develop aims to communicate two people through different mechanisms:

- **Text messages.**
- **Images.**
- **Videos.**
- **Video calls.**

However, the approach that the application will take it´s around permits. A user can not send a file of any kind or speak with a person if they do not have explicit permission from the recipient.

But the most important point of the chat is not the permissions, but the decentralization. We want only users to have access to the content of those conversations, so that they are the owners of their personal information.

To achieve this goal we will use SOLID. SOLID is a project that consists of a decentralized website. It deals with establishing a connection between several users, separating data application. To establish this connection, we will use what is known as POD. A POD is a unique profile of each user, which allows to store information on the web about it.

We will use these PODs to connect both users, but the data they exchange will be stored in the interlocutors' devices. In this way, we will create a private chat that will have the advantage of being isolated, reliable and safe.

```
=== Business Context
```

- Communication elements:
  - **Users:** Users are the main recipient of the application.
  - **Web:** Users will connect to the web through their PODs to establish secure communication.
- Communication process:
  - The partners create a POD.
  - The user adds as friend through the PODs to the second interlocutor.
  - The interlocutors log in with their POD in the chat.
  - Ready!

What data are exchanged with the environment? Through the website, users will enter some small data on the SOLID website to obtain their POD. Once established the connection between both interlocutors, they will be the only owners and holders of the information they exchange.

Communication Partner	Input	Output
User	Chat, video, images, POD information	Chat, video, images, POD information
Web		POD, Server Connection

=== Technical Context

As we have said before, we will use the so-called PODs provided by SOLID to connect to the server of the same, and through it, communicate with other people.

For the design of the connection we will use different mechanisms:

- **RDF:** This library allows us to establish the connection. Act as a communicator and thanks to it we can obtain the POD information necessary to establish communication.
- **Authenticator:** This tool receives the POD of a user and allows us to know if it is a valid POD.
- **Angular:** It is a JavaScript framework that will allow us to handle the language in a more comfortable way.

[03 System Scope and Context] | *images/03_System_Scope_and_Context.JPG*

== Solution Strategy

=== Technology and IDE The team had to choose between betweet new technologies for us and technologies already kown but that dd no have as much suport to SOLID. Based on what we knew, our first option was to choose Java as programming language, but after reading the SOLID documentation, we decided to choose JavaScript, more specifically, the Angular Framework.

About the IDE, we choose WebStorm due to its power and the fact that, as students, we have a free payment license. In addition, some of the members of the group have used it in previous subjects.

=== Architectural pattern We decided to use the architectural pattern model-view-controller (MVC) because it is a pattern that gives us more extensibility and maintainability. Also, its implementation supports rapid and parallel development. For more information visit: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

=== Methodology of work We will use Scrum to develope this project. Scrum can provide us an early software development, and more independence when programming. Our project will start implementing the basic functions of a chat and it will grow in complexity (https://en.wikipedia.org/wiki/Scrum_(software_development))

```
== Building Block View
```

Content

The building block view shows the static decomposition of the system into building blocks (modules, components, subsystems, classes, interfaces, packages, libraries, frameworks, layers, partitions, tiers, functions, macros, operations, datas structures, ...) as well as their dependencies (relationships, associations, ...)

This view is mandatory for every architecture documentation. In analogy to a house this is the *floor plan*.

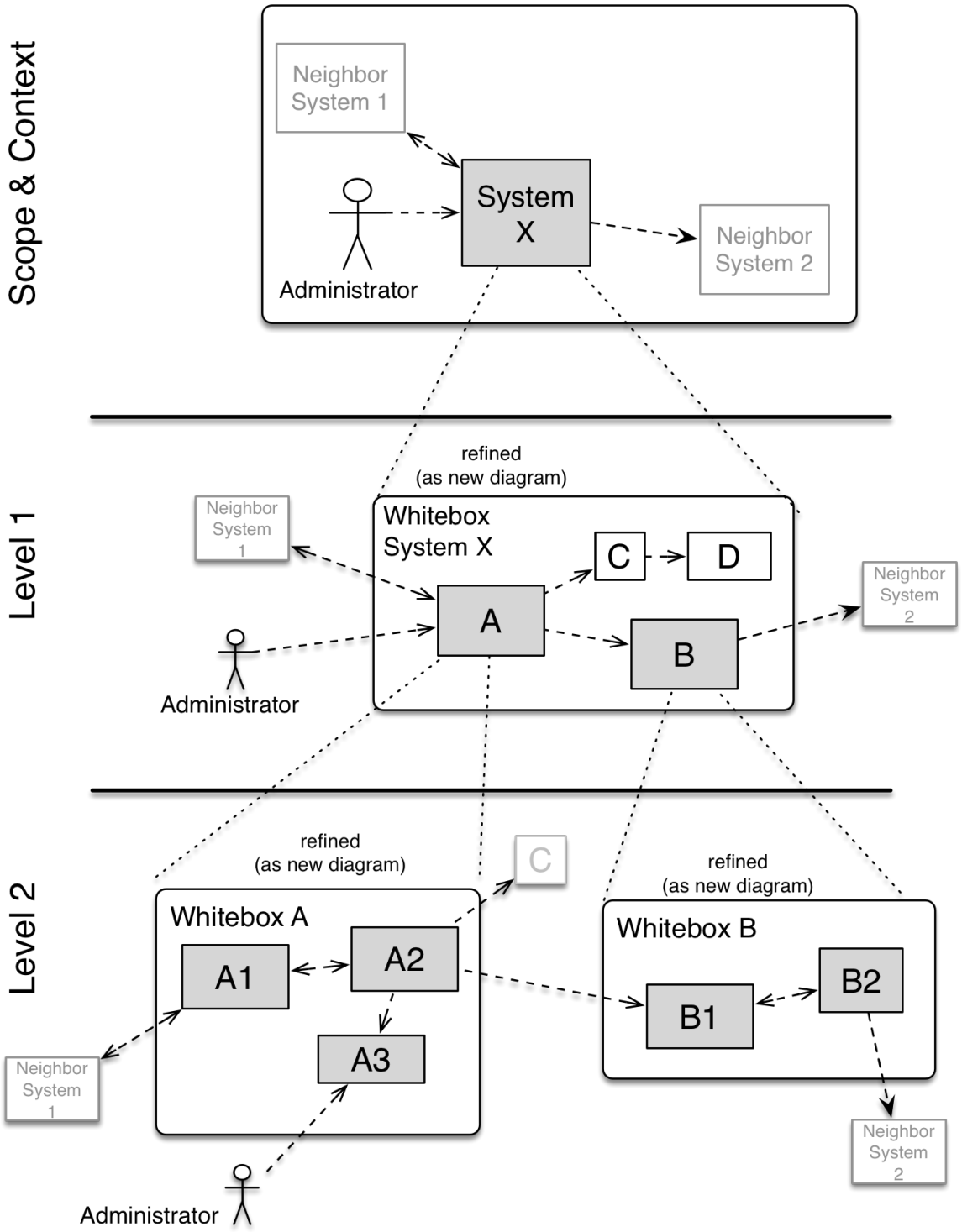
Motivation

Maintain an overview of your source code by making its structure understandable through abstraction.

This allows you to communicate with your stakeholder on an abstract level without disclosing implementation details.

Form

The building block view is a hierarchical collection of black boxes and white boxes (see figure below) and their descriptions.



Level 1 is the white box description of the overall system together with black box descriptions of all contained building blocks.

Level 2 zooms into some building blocks of level 1. Thus it contains the white box description of selected building blocks of level 1, together with black box descriptions of their internal building blocks.

Level 3 zooms into selected building blocks of level 2, and so on.

```
=== Whitebox Overall System
```

Here you describe the decomposition of the overall system using the following white box template. It contains

- an overview diagram
- a motivation for the decomposition
- black box descriptions of the contained building blocks. For these we offer you alternatives:
  - use *one* table for a short and pragmatic overview of all contained building blocks and their interfaces
  - use a list of black box descriptions of the building blocks according to the black box template (see below). Depending on your choice of tool this list could be sub-chapters (in text files), sub-pages (in a Wiki) or nested elements (in a modeling tool).
- (optional:) important interfaces, that are not explained in the black box templates of a building block, but are very important for understanding the white box. Since there are so many ways to specify interfaces why do not provide a specific template for them. In the worst case you have to specify and describe syntax, semantics, protocols, error handling, restrictions, versions, qualities, necessary compatibilities and many things more. In the best case you will get away with examples or simple signatures.

> *<**Overview Diagram**>*
>
> **Motivation**
>
> *<text explanation>*
>
> **Contained Building Blocks**
>
> *<Description of contained building block (black boxes)>*
>
> **Important Interfaces**
>
> *<Description of important interfaces>*

Insert your explanations of black boxes from level 1:

If you use tabular form you will only describe your black boxes with name and responsibility according to the following schema:

Name	**Responsibility**
<black box 1>	*<Text>*
<black box 2>	*<Text>*

If you use a list of black box descriptions then you fill in a separate black box template for every important building block . Its headline is the name of the black box.

```
==== <Name black box 1>
```

Here you describe <black box 1> according the the following black box template:

- Purpose/Responsibility
- Interface(s), when they are not extracted as separate paragraphs. This interfaces may include qualities and performance characteristics.
- (Optional) Quality-/Performance characteristics of the black box, e.g.availability, run time behavior, ….
- (Optional) directory/file location
- (Optional) Fulfilled requirements (if you need traceability to requirements).
- (Optional) Open issues/problems/risks

```
<Purpose/Responsibility>

<Interface(s)>

<(Optional) Quality/Performance Characteristics>

<(Optional) Directory/File Location>

<(Optional) Fulfilled Requirements>

<(optional) Open Issues/Problems/Risks>

==== <Name black box 2>

<black box template>

==== <Name black box n>

<black box template>

==== <Name interface 1>

…

==== <Name interface m>

=== Level 2
```

Here you can specify the inner structure of (some) building blocks from level 1 as white boxes.

You have to decide which building blocks of your system are important enough to justify such a detailed description. Please prefer relevance over completeness. Specify important, surprising, risky, complex or volatile building blocks. Leave out normal, simple, boring or standardized parts of your system

```
==== White Box <building block 1>
```

…describes the internal structure of *building block 1*.

<*white box template*>

==== White Box <*building block 2*>

<*white box template*>

...

==== White Box <*building block m*>

<*white box template*>

=== Level 3

Here you can specify the inner structure of (some) building blocks from level 2 as white boxes.

When you need more detailed levels of your architecture please copy this part of arc42 for additional levels.

==== White Box <_building block x.1_>

Specifies the internal structure of *building block x.1*.

<*white box template*>

==== White Box <_building block x.2_>

<*white box template*>

==== White Box <_building block y.1_>

<*white box template*>

```
== Runtime View
```

Contents

The runtime view describes concrete behavior and interactions of the system's building blocks in form of scenarios from the following areas:

- important use cases or features: how do building blocks execute them?
- interactions at critical external interfaces: how do building blocks cooperate with users and neighboring systems?
- operation and administration: launch, start-up, stop
- error and exception scenarios

Remark: The main criterion for the choice of possible scenarios (sequences, workflows) is their **architectural relevance**. It is **not** important to describe a large number of scenarios. You should rather document a representative selection.

Motivation

You should understand how (instances of) building blocks of your system perform their job and communicate at runtime. You will mainly capture scenarios in your documentation to communicate your architecture to stakeholders that are less willing or able to read and understand the static models (building block view, deployment view).

Form

There are many notations for describing scenarios, e.g.

- numbered list of steps (in natural language)
- activity diagrams or flow charts
- sequence diagrams
- BPMN or EPCs (event process chains)
- state machines
- …

=== <Runtime Scenario 1>

- *<insert runtime diagram or textual description of the scenario>*
- *<insert description of the notable aspects of the interactions between the building block instances depicted in this diagram.>*

=== <Runtime Scenario 2>

=== …

=== <Runtime Scenario n>

```
== Deployment View
```

Content

The deployment view describes:

1. the technical infrastructure used to execute your system, with infrastructure elements like geographical locations, environments, computers, processors, channels and net topologies as well as other infrastructure elements and
2. the mapping of (software) building blocks to that infrastructure elements.

Often systems are executed in different environments, e.g. development environment, test environment, production environment. In such cases you should document all relevant environments.

Especially document the deployment view when your software is executed as distributed system with more then one computer, processor, server or container or when you design and construct your own hardware processors and chips.

From a software perspective it is sufficient to capture those elements of the infrastructure that are needed to show the deployment of your building blocks. Hardware architects can go beyond that and describe the infrastructure to any level of detail they need to capture.

Motivation

Software does not run without hardware. This underlying infrastructure can and will influence your system and/or some cross-cutting concepts. Therefore, you need to know the infrastructure.

Form

Maybe the highest level deployment diagram is already contained in section 3.2. as technical context with your own infrastructure as ONE black box. In this section you will zoom into this black box using additional deployment diagrams:

- UML offers deployment diagrams to express that view. Use it, probably with nested diagrams, when your infrastructure is more complex.
- When your (hardware) stakeholders prefer other kinds of diagrams rather than the deployment diagram, let them use any kind that is able to show nodes and channels of the infrastructure.

```
=== Infrastructure Level 1
```

Describe (usually in a combination of diagrams, tables, and text):

- the distribution of your system to multiple locations, environments, computers, processors, .. as well as the physical connections between them
- important justification or motivation for this deployment structure
- Quality and/or performance features of the infrastructure
- the mapping of software artifacts to elements of the infrastructure

For multiple environments or alternative deployments please copy that section of arc42 for all relevant environments.

> *<Overview Diagram>*
>
> **Motivation**
>
> *<explanation in text form>*
>
> **Quality and/or Performance Features**
>
> *<explanation in text form>*
>
> **Mapping of Building Blocks to Infrastructure**
>
> *<description of the mapping>*
>
> === Infrastructure Level 2

Here you can include the internal structure of (some) infrastructure elements from level 1.

Please copy the structure from level 1 for each selected element.

==== <*Infrastructure Element 1*>

<*diagram + explanation*>

==== <*Infrastructure Element 2*>

<*diagram + explanation*>

…

==== <*Infrastructure Element n*>

<*diagram + explanation*>

```
== Cross-cutting Concepts
```

Content

This section describes overall, principal regulations and solution ideas that are relevant in multiple parts (= cross-cutting) of your system. Such concepts are often related to multiple building blocks. They can include many different topics, such as

- domain models
- architecture patterns or design patterns
- rules for using specific technology
- principal, often technical decisions of overall decisions
- implementation rules

Motivation

Concepts form the basis for *conceptual integrity* (consistency, homogeneity) of the architecture. Thus, they are an important contribution to achieve inner qualities of your system.

Some of these concepts cannot be assigned to individual building blocks (e.g. security or safety). This is the place in the template that we provided for a cohesive specification of such concepts.

Form

The form can be varied:

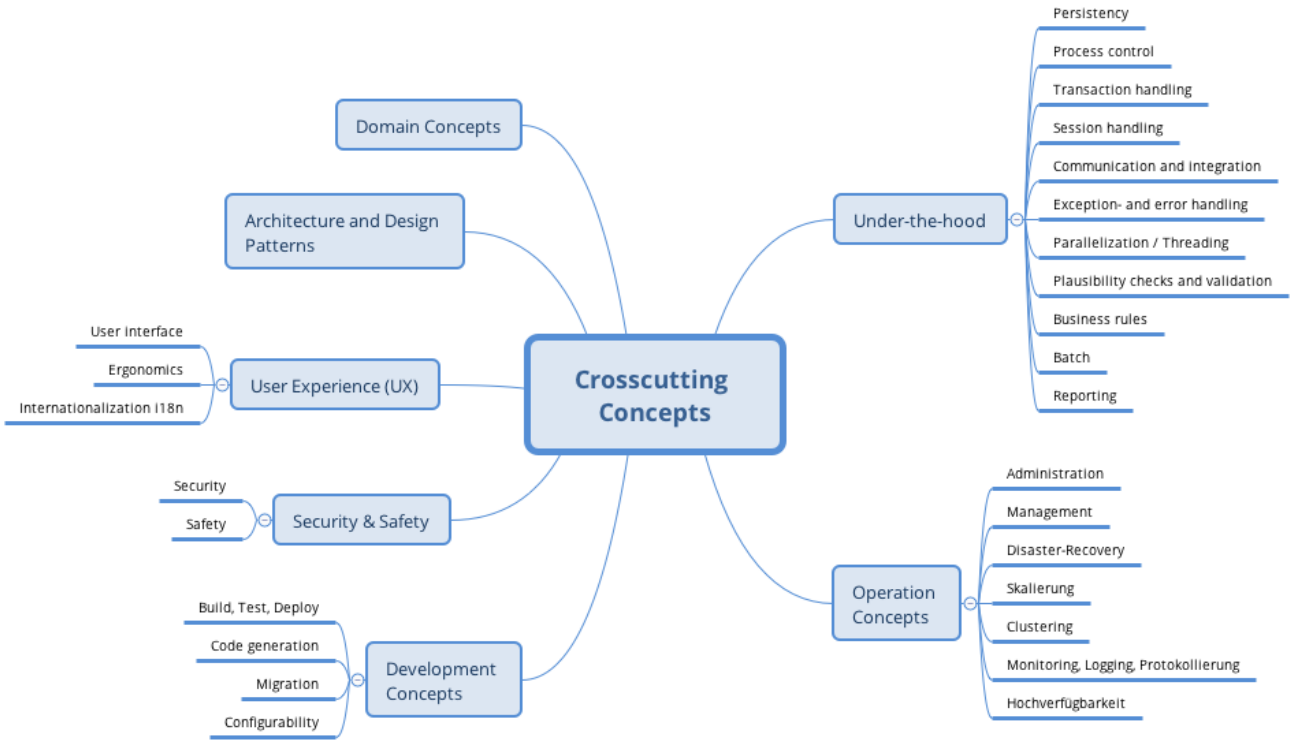
- concept papers with any kind of structure
- cross-cutting model excerpts or scenarios using notations of the architecture views
- sample implementations, especially for technical concepts
- reference to typical usage of standard frameworks (e.g. using Hibernate for object/relational mapping)

Structure

A potential (but not mandatory) structure for this section could be:

- Domain concepts
- User Experience concepts (UX)
- Safety and security concepts
- Architecture and design patterns
- "Under-the-hood"
- development concepts
- operational concepts

Note: it might be difficult to assign individual concepts to one specific topic on this list.


Crosscutting Concepts
Domain Concepts
Architecture and Design Patterns
User Experience (UX)
User Interface
Ergonomics
Internationalization i18n
Security & Safety
Security
Safety
Development Concepts
Build, Test, Deploy
Code generation
Migration
Configurability
Under-the-hood
Persistency
Process control
Transaction handling
Session handling
Communication and integration
Exception- and error handling
Parallelization / Threading
Plausibility checks and validation
Business rules
Batch
Reporting
Operation Concepts
Administration
Management
Disaster-Recovery
Skalierung
Clustering
Monitoring, Logging, Protokollierung
Hochverfügbarkeit

=== <*Concept 1*>

<*explanation*>

=== <*Concept 2*>

<*explanation*>

…

=== <*Concept n*>

<*explanation*>

== Design Decisions

- First decision is that we are using Git as version control system instead of using mercurial, vsn, etc. and GitHub as source code management instead of using Bitbucket, this decision was imposed by the teachers of the subject.
- We are using javascript family language(javascript, typescript, node.js, angular, etc) to build the application as the main programming language. As we understand, javascript is the easiest way to make a SOLID application.
- We are also using basic languages for the web such as html, css, etc.
- Social linked data(SOLID) based application. A must decision we have to follow if we want to make a decentralized application. This means following several important constraints to build the application. One of them is storing date in PODS, so the user is the owner of their data.
- Following MVC architecture as much as we can. It is the main architecture model we know something about, so we are going to try to follow it as much as we can.
- Following Angular components structure, so we can build a high cohesion and low coupling app.
- To be continued.

===== TODO Improve Mind-Map ===== TODO Full decisions table?

```
== Quality Requirements
```

Content

This section contains all quality requirements as quality tree with scenarios. The most important ones have already been described in section 1.2. (quality goals)

Here you can also capture quality requirements with lesser priority, which will not create high risks when they are not fully achieved.

Motivation

Since quality requirements will have a lot of influence on architectural decisions you should know for every stakeholder what is really important to them, concrete and measurable.

```
=== Quality Tree
```

Content

The quality tree (as defined in ATAM – Architecture Tradeoff Analysis Method) with quality/evaluation scenarios as leafs.

Motivation

The tree structure with priorities provides an overview for a sometimes large number of quality requirements.

Form

The quality tree is a high-level overview of the quality goals and requirements:

- tree-like refinement of the term "quality". Use "quality" or "usefulness" as a root
- a mind map with quality categories as main branches

In any case the tree should include links to the scenarios of the following section.

```
=== Quality Scenarios
```

Contents

Concretization of (sometimes vague or implicit) quality requirements using (quality) scenarios.

These scenarios describe what should happen when a stimulus arrives at the system.

For architects, two kinds of scenarios are important:

- Usage scenarios (also called application scenarios or use case scenarios) describe the system’s runtime reaction to a certain stimulus. This also includes scenarios that describe the system’s efficiency or performance. Example: The system reacts to a user’s request within one second.
- Change scenarios describe a modification of the system or of its immediate environment. Example: Additional functionality is implemented or requirements for a quality attribute

change.

Motivation

Scenarios make quality requirements concrete and allow to more easily measure or decide whether they are fulfilled.

Especially when you want to assess your architecture using methods like ATAM you need to describe your quality goals (from section 1.2) more precisely down to a level of scenarios that can be discussed and evaluated.

Form

Tabular or free form text.

```
== Risks and Technical Debts
```

Contents

A list of identified technical risks or technical debts, ordered by priority

Motivation

“Risk management is project management for grown-ups” (Tim Lister, Atlantic Systems Guild.)

This should be your motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g. project managers, product owners) as part of the overall risk analysis and measurement planning.

Form

List of risks and/or technical debts, probably including suggested measures to minimize, mitigate or avoid risks or reduce technical debts.

Risks	**Solutions**
Lack of knowledge and use of angularjs	• Consult forums • Look at the documentation • take courses
Lack of knowledge of solid	• Look at the solid forums • Ask other classmates
Abandonment / lack of work of a member	• Know the status of the entire project • Hold periodic meetings
Lack of time to complete the project	• Do it stepwise and continue
Misunderstood the requirements of the application	• Consult with the professor about the requirements
Problems with git	• Use a specific methodology to do merges and commits

```
== Glossary
```

Definition of most important terms every stakeholder should know to understand the documentation.

Order by rarety.

Only in english at the moment.

Term	Definition
Decentralized	User is the owner of their data and not the application
Git	Version control system
Inrupt	SOLID enterprise
MVC	Model-View-Controler, software architecture
PODS	Personal Online Data Stores, used to save data
RDF	Resource Description Framework, standard model for data interchange in the web
Scrum	Agile methodology for working in group
SOLID	Social Linked Data
Stakeholders	People interested in a proyect

About arc42

arc42, the Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 7.0 EN (based on asciidoc), January 2017