UhuBank - Documento de Arquitetura de Software

table of contents

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

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

1.1. Requirements Overview

O principal propósito da evolução do sistema bancário da Fintech UhuBank é poder ofertar novos serviços bancários, além do cartão de crédito pré-pago já oferecido, integrando-se há um banco de grande porte, através de uma Open Banking API.

Os novos serviços que serão oferecidos são:

- Conta Corrente Digital
- Cartão de Crédito
- Empréstimo

Todos os serviços devem ser digitais, contactless, ou seja, o cliente tem a opção de realizar 100% das interações via canais digitais.

1.2. Quality Goals

Nr.	Quality	Motivation
1	Availability	Verificar legislação, se não houver, manter arbitrariamente a disponibilidade em 99,99%. Deve-se minimizar o máximo possível de downtime.
2	Interoperability	A aplicação deve prover uma API para uso das lojas virtuais com o objetivo de verificar crédito e processar o pagamento de uma determinada venda. Deve-se também criar uma camada cliente para comunicação com a Open Banking API.
3	Time Behaviour	Processos como os de conta corrente digital, extrato e contações de moeda devem acontecer em tempo real.
4	Testability	A arquitetura deve permitir a facilidade nos testes dos principais blocos de construção.
5	Non-repudiation	Quaisquer operações e/ou eventos devem ser registrados para fins de auditoria para que os mesmos não possam ser repudiados posteriormente.
6	Fault Tolerance	O sistema deve ser resiliente e escalável horizontalmente e verticalmente afim de manter a qualidade de acesso aos serviços.

1.3. Stakeholders

Role/Name	Goal/Boundaries
CEO Hari Dinesh	É o principal Product Owner. Seu objetivo é assegurar que todas as mudanças podem ser feitas de forma que consiga manter a qualidade do aplicativo/sistema web para os usuários finais.
Developers	Desenvolvedores que querem informações como persistir e recuperar dados do banco de dados Datomic. Informações sobre os principais design patterns que serão utilizados na arquitetura de microserviços.
Analistas de configuração e de infraestrutura	Necessitam saber como os sistemas serão implantados e configurados.

2. Architecture Constraints

2.1. Technical Constraints

Constraints	Background and/or Motivation
Implementação	O frontend deve ser implementado utilizando o framework Javascript Angular. Os microserviços devem ser implementados utilizando as seguintes linguagens e frameworks: Java 8 de preferência; Clojure para persistência no Datomic; Python e Scala para Big Data, Machine Learning, Deep Learning e Analytics; Spring Boot e Spring Cloud.
Banco de Dados	Datomic e Amazon DynamoDB
Bibliotecas e Componentes	Todas as dependências externas devem estar disponibilizadas através de um package manager. Devem também, estar sob licença open source compatível.
Implantação	Os serviços devem ser implantados em containers Docker.

2.2. Organization Constraints

Constraint	Background and/or Motivation
Configuration and version control management	Cada serviço deve ter seu próprio repositório. Todos os repositórios são privados e pertencem a organização. Os commits e a organização das branchs devem seguir o processo GitFlow.
Testing	Utilizar Junit, Mockito e Hamcrest para criação de testes unitários e de integração. Utilizar Cucumber para os critérios de aceitação, e Gatling para os testes de performance. No Frontend deve-se utilizar Jasmine e Karma.

2.3. Conventions

Conventions	**Background and/or Motivation**
Documentação de Arquitetura	Baseado na estrutura do template arc42 em inglês na versão 7.0
Representação Arquitetural	A arquitetura será apresentada de acordo com um modelo conhecido como C4. O modelo C4 é uma abordagem que utiliza abstrações para a diagramação da arquitetura de software. Esse modelo provê uma maneira de comunicar eficientemente a arquitetura de software, em diferentes níveis de detalhes, para públicos distintos. O modelo C4 consiste em um conjunto hierárquico de diagramas de arquitetura de software para contexto, containers, componentes e código.
Code Conventions	O projeto utiliza convenções da linguagem de programação Java e são verificadas através do Checkstyle e/ou DocConfig. Deve-se procurar as conveções e/ou padrões das outras linguagens permitidas no projeto.
Execução das Builds	Todas as builds, exceto as executadas no ambiente local de desenvolvimento, devem rodar com o profile **prod**.
Idioma	O inglês deve ser utilizado em todo o projeto.

3. System Scope and Context

> O diagrama de contexto do sistema permite ter uma visão geral. Os detalhes não são importantes. O foco deve ser nos usuários e nos sistemas de software que se conectam ou interagem diretamente ao sistema.

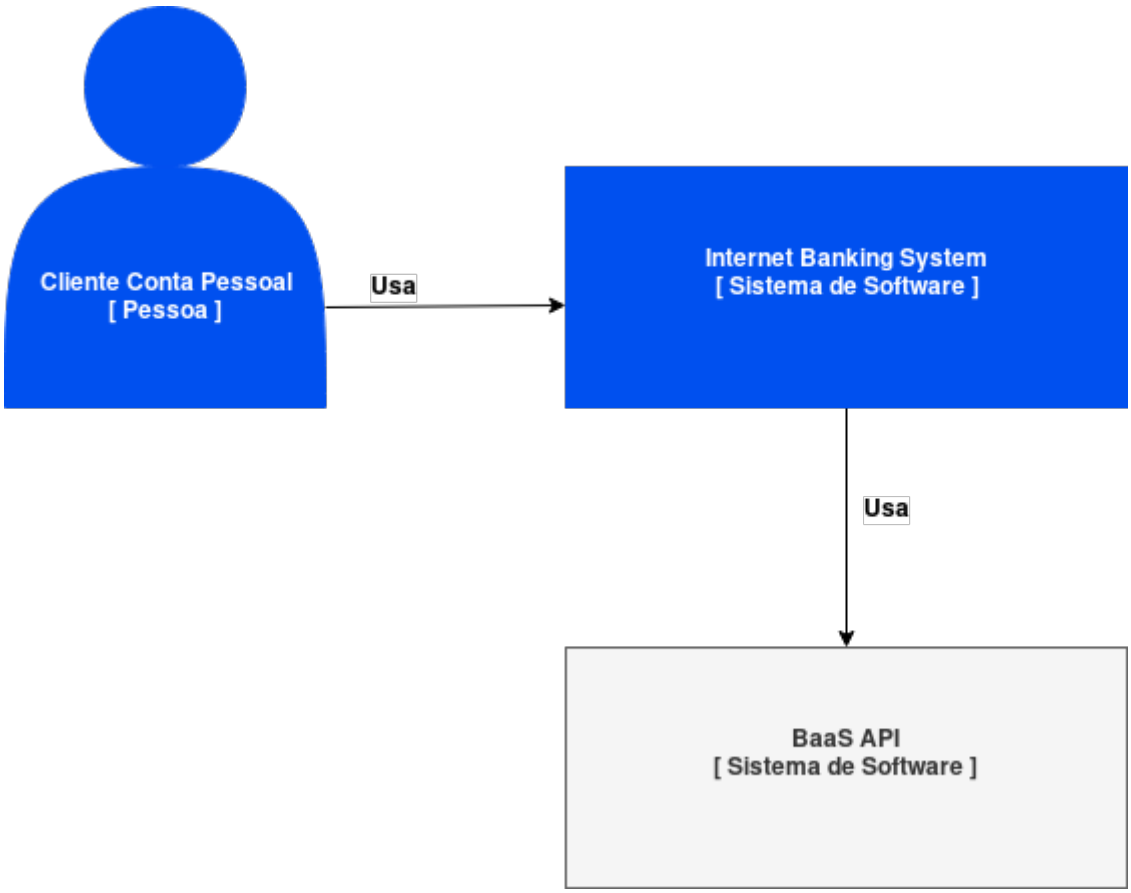


Figure 1. System Context Diagram

O contexto do negócio foi baseado e levou em consideração o modelo Open Banking, onde um determinado banco expõe os seus sistemas através de APIs e padrões abertos, permitindo que terceiros (outros bancos, fintechs, insuretechs, desenvolvedores independentes, etc) ofereçam novos serviços e experiências para o cliente final.

4. Solution Strategy

Contents

A short summary and explanation of the fundamental decisions and solution strategies, that shape the system's architecture. These include

- technology decisions
- decisions about the top-level decomposition of the system, e.g. usage of an architectural pattern or design pattern
- decisions on how to achieve key quality goals
- relevant organizational decisions, e.g. selecting a development process or delegating certain tasks to third parties.

Motivation

These decisions form the cornerstones for your architecture. They are the basis for many other detailed decisions or implementation rules.

Form

Keep the explanation of these key decisions short.

Motivate what you have decided and why you decided that way, based upon your problem statement, the quality goals and key constraints. Refer to details in the following sections.

5. 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*.

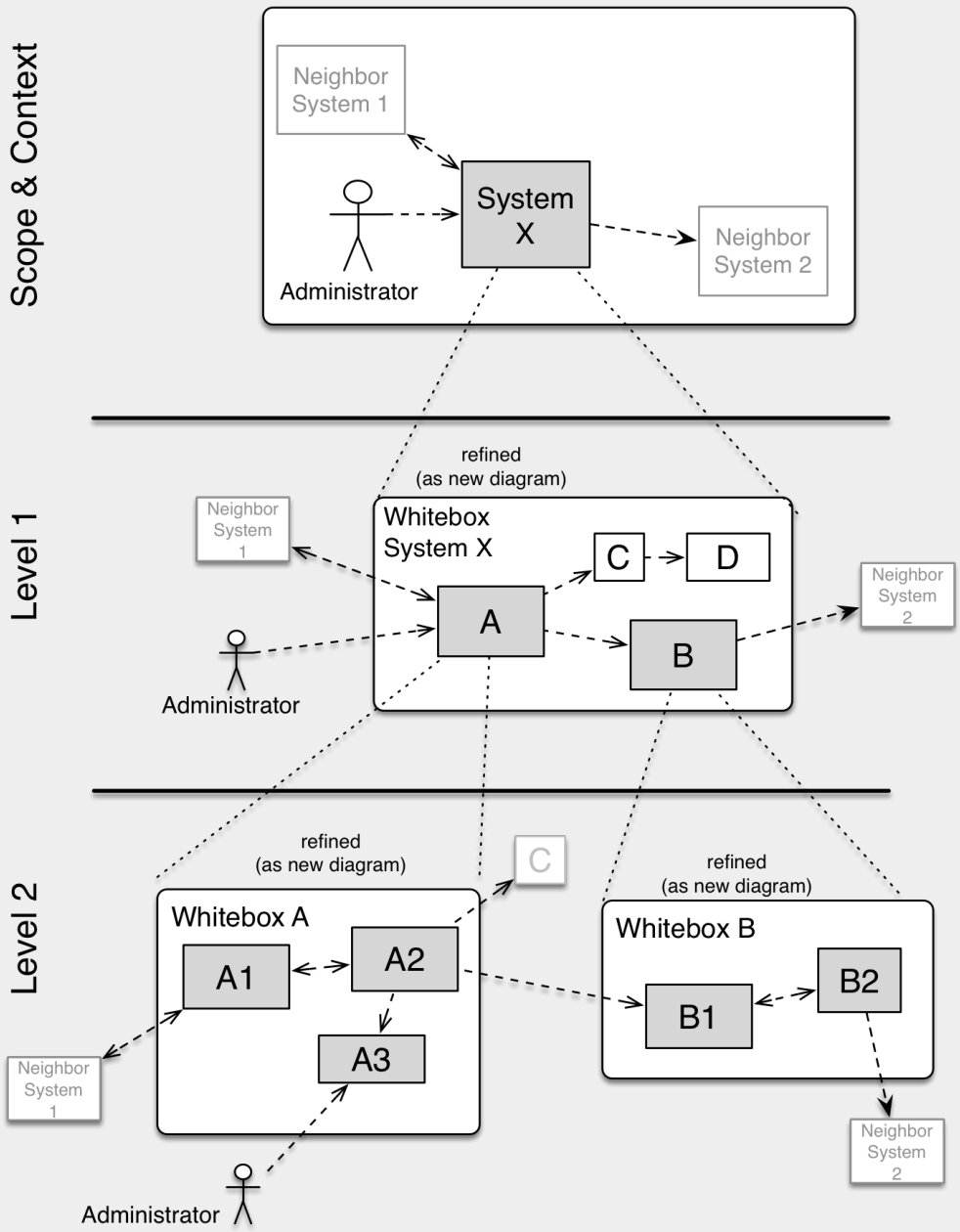
Motivation

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.

5.1. 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.

5.1.1. <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>

5.1.2. <Name black box 2>

<black box template>

5.1.3. <Name black box n>

<black box template>

5.1.4. <Name interface 1>

...

5.1.5. <Name interface m>

5.2. 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

5.2.1. White Box *<building block 1>*

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

<white box template>

5.2.2. White Box *<building block 2>*

<white box template>

...

5.2.3. White Box *<building block m>*

<white box template>

5.3. 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.

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

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

<white box template>

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

<white box template>

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

<white box template>

6. 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
- ...

6.1. <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.>*

6.2. <Runtime Scenario 2>

6.3. ...

6.4. <Runtime Scenario n>

7. 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.

7.1. 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>

7.2. 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.

7.2.1. *<Infrastructure Element 1>*

<diagram + explanation>

7.2.2. *<Infrastructure Element 2>*

<diagram + explanation>

...

7.2.3. *<Infrastructure Element n>*

<diagram + explanation>

8. 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:

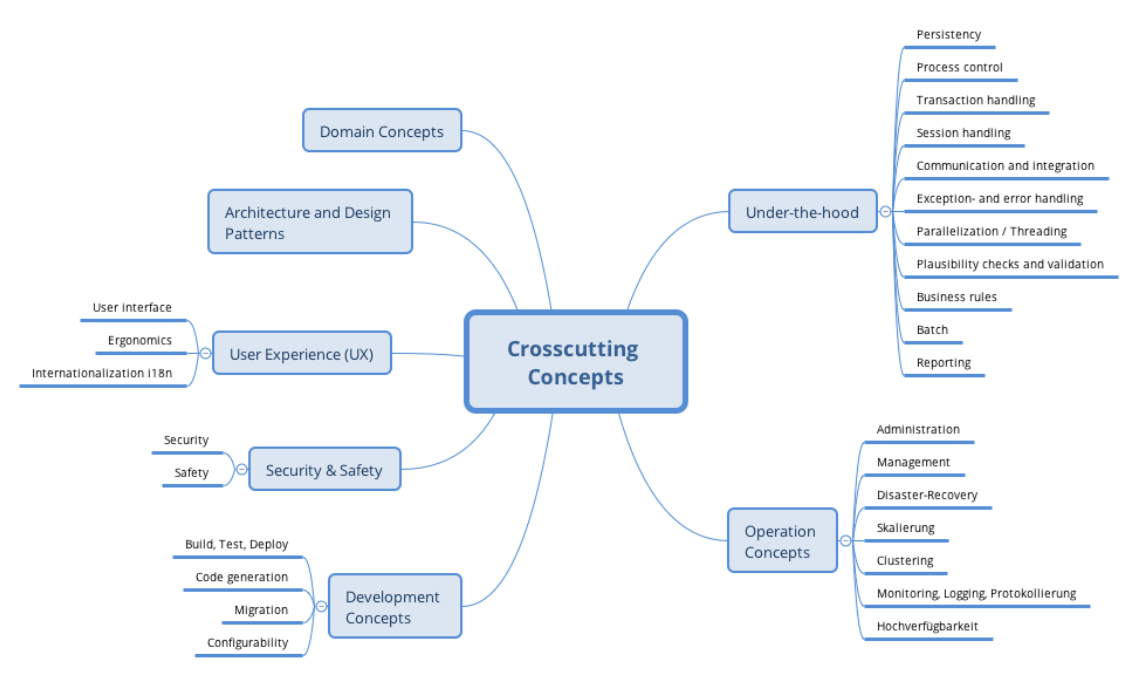
- concept papers with any kind of structure
- cross-cutting model excerpts or scenarios using notations of the architecture views
- sample implementations, especially for technical concepts
- reference to typical usage of standard frameworks (e.g. using Hibernate for object/relational mapping)

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.



8.1. *<Concept 1>*

<explanation>

8.2. *<Concept 2>*

<explanation>

...

8.3. *<Concept n>*

<explanation>

9. Design Decisions

9.1. Escolha do Datomic e do Amazon DynamoDB como Banco de Dados

Problema

Os serviços lidarão com massivos volumes de dados. Algumas consultas serão complexas. Há necessidade de se prover facilidades para Analytics. O controle transacional é muito importante. O banco de dados precisa ser escalável.

Alternativas Consideradas

- Utilizar banco de dados relacional convencional. O problema é que os bancos de dados relacionais são progressivamente lentos e difíceis de escalar quando se tem um aumento substancial na volumetria de dados.
- Utilizar Datomic e Amazon DynamoDB.

Decisão

Eu optei por utilizar Datomic e Amazon DynamoDB como banco de dados. Datomic é um banco de dados diferente dos bancos tradicionais.

Datomic armazena os dados diretamente em múltiplos índices. Ele considera dados e índices como sendo a mesma coisa.

Datomic é um banco de dados em cima de outro banco de dados. Ele não escreve diretamente no file system. Ao invés disso ele utiliza outro banco de dados como storage backend. Nessa cenário, o Amazon DynamoDB foi escolhido como storage backend porque os dois já tem uma integração bastante consolidada.

Datomic não é monolítico igual aos outros bancos de dados tradicionais. Ele possui diferentes APIs para leitura e escrita. Datomic utiliza um "transactor" API para escrita e Datalog para leitura. Isso facilita a implementação do microservice pattern CQRS.

Datomic é um banco de dados imutável. O conceito é parecido ao sistema de controle de versões, como o Git. Os dados nunca são sobrescritos ou atualizados. Você apenas marca o dado inválido e adiciona um novo dado.

10. 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.

10.1. 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.

10.2. 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.

11. Risks and Technical Debts

Risco 1

Datomic é um banco de dados complementamente diferente dos bancos de dados tradicionais e desconhecido da grande maioria dos desenvolvedores e é relativamente novo. A curva de aprendizado é um pouco alta. Esse risco pode ser mitigado através de treinamentos, consultoria, workshops, disponibilização de materiais de estudos e mentoring.

Risco 2

Em relação a Open Banking API, há um grande risco no Brasil. No momento ainda não não há regulamentação. O Banco Central já começou as discussões com o mercado, afim de coordenar a definição desses padrões. A programação é que até o final de 2019 o modelo seja implementado. Em abril deste ano (2018), o Banco Central regulamentou a atuação das fintechs de crédito. A idéia agora é começar a implementação dos serviços que já são regulamentados.

12. Glossary

Termo	Definição
Endpoint	Ponto de entrada dos serviços REST. Responsável por receber as requisições dos serviços.
REST	A Representational State Transfer, em português, Transferência de Estado Representacional. É uma abstração da arquitetura da World Wide Web, ou seja, um estilo arquitetural que consiste de um conjunto coordenado de restrições arquiteturais aplicadas componentes, conectores e elementos de dados dentro de um sistema de hipermídia distribuído. O REST ignora os detalhes de implementação de componente e a sintaxe de protocolo com o objetivo de focar nos papéis dos componentes, nas restrições sobre sua interação com outros componentes e na sua interpretação de elementos de dados significantes.
Hibernate	É um framework para o mapeamento objeto-relacional escrito na linguagem Java. O objetivo do Hibernate é diminuir a complexidade entre os programas Java orientados a objetos e os bancos de dados ditos modelo relacional, em especial, no desenvolvimento de consultas e atualizações dos dados. Sua principal característica é a transformação das classes em Java para tabelas de dados e dos tipos de dados Java para os da SQL. O Hibernate gera as consultas SQL e libera o desenvolvedor do trabalho manual da conversão dos dados resultantes, mantendo o programa portável para quaisquer bancos de dados.
JPA	É o padrão definido para a linguagem Java que descreve uma interface comum para frameworks de persistência de dados. A JPA define uma meio de mapeamento objeto-relacional para os objetos Java simples e comuns (POJOs), denominados de entidade.
WSDL	Web Services Description Language. É uma linguagem baseada em XML utilizada para descrever Web Services funcionando como um contrato do serviço. Trata-se de um documento escrito em XML que além de descrever o serviço, especifica como acessá-lo e quais são as operações ou métodos disponíveis.

Termo	Definição
BEAN	Objetos Java gerenciados pelo container do Spring. Esses beans são criados, instanciados a partir dos metadados de configuração fornecidos ao container, como anotações e arquivos de configurações.
Singleton	Padrão de projeto que define uma única instância de um objeto por usuário ou aplicação.
Kubernetes	É um sistema open-source de automação de implantação, escalabilidade e gereciamento de containers através de clusters de hosts.