InfluenzaNet 2.0: Architecture Document (version 1.3)

Based on Master's Thesis: "Design and Implementation of a Scalable Distributed Data Collection Architecture for a European Wide Flu Monitoring System" by Maša Reko
Date: last updated 9.1.2019
More information: https://influenzanet.github.io/

Document status: _ Draft __ Proposed __ Validated _X_ Approved

1. Introduction

This document contains the system architecture for the Master thesis titled "***Design and Implementation of a Scalable Distributed Data Collection Architecture for a European Wide Flu Monitoring System***" and serves as basis for the Influenzanet 2.0 development.

1.1 Scope and Summary

The purpose of this document is to give an overview of the architecture for the InfluenzaNet backend application. The architecture is described through four views (see Section 4): logical, process, physical and development view. The structure of each view is explained in Section 2. All views describe the same architecture, but some views may be more interesting to the reader than others, depending on his/her role in the project. For guidance on how to read and use this document, please refer to Section 1.4.

It should be noted that this document is closely related to the InfluenzaNet Requirements Document. The architecture relies heavily on the functional and quality requirements explained in the requirements document.

1.2 Overview Sections

Section 1 explains the scope and structure of this document.

Section 2 explains the organization of each view that can be found in this document.

Section 3 gives an overview of the general architecture style used for the system.

Section 4 gives a detailed description of all the views in the format explained in Section 2.

Section 5 provides the tables explaining the relationships between the views.

Section 6 explains the origins of architectural decisions that apply to more than one view, as well as the influence of the requirements on the architecture of this system.

Section 7 contains reference material, namely a list of glossary terms.

1.3 View Overview

As outlined above, this document describes four different views of the architecture. Below is the summary of those views. For a more detailed description, see Section 4.

Name	Logical View
Architectural patterns	Microservices
Design patterns	Decompose by Business Capability
Element types	Classes
Relation types	Dependency, Association
Modeling techniques	UML (Class diagram)

Name	Process View
Architectural patterns	Microservices
Design patterns	Database per service, API Gateway, Circuit Breaker, Access Token
Element types	Objects, Roles, Databases, Browsers
Relation types	Messages, Replies
Modeling techniques	UML (Interaction diagram)

Name	Physical View
Architectural patterns	Microservices
Design patterns	Database per service, API Gateway, Service Instance Per Container
Element types	Nodes, Artifacts
Relation types	Association
Modeling techniques	UML (Deployment diagram)

Name	Development View
Architectural patterns	Microservices
Design patterns	Decompose by Business Capability
Element types	Packages, Classes
Relation types	Dependency
Modeling techniques	UML (Package diagram)

1.4 How to Use This Document

The table below shows how various stakeholders might use this document to help address their concerns. Some views may be more useful than others to a certain stakeholder, and that is why this table may be helpful in finding exactly what the reader is looking for.

Stakeholder	Logical View	Process View	Physical View	Development View
User	explains what the system does and how	/	/	/
Acquirer	explains what the system does and how	/	/	/
Developer	explains what the system does and how	explains how parts of the system exchange information	/	explains how parts of software are related and dependent on one another
Maintainer	explains what the system does and how	explains how parts of the system exchange information	/	explains how parts of software are related and dependent on one another
Tester	explains what the system does and how	explains how parts of the system exchange information	/	/
Integrator	explains what the system does and how	explains how parts of the system exchange information	/	explains how parts of software are related and dependent on one another
Network administrator	explains what the system does and how	/	explains where the packages fit on the various physical parts of the system	/

2. How a View is Documented

Each view is organized in three sections, as follows:

1. *Primary Presentation*

This section shows the elements and relations of the view, in a graphical form, using the UML notation. The main goal of the section is to provide an introduction with a short summary of the view in question.

2. *Element Catalog*

This section provides the details of the elements from the primary presentation, as well as those elements and relations that were omitted from it, but that are also relevant to the view.

It is divided into four sub-sections:

a) *Elements and their properties.* This sub-section gives a list of all elements in the view, along with their properties.
b) *Relations and their properties.* This sub-section gives a list of all relations between the elements, along with their properties. All possible exceptions to what is shown in the primary presentation are also included.
c) *Element interfaces*. This sub-section documents all element interfaces.
d) *Element behavior.* This sub-section explains the element behavior if it is not easily understood from the primary presentation.

3. Rationale

This section explains the origin of the design reflected in the view. In other words, this section shows the importance of the design being the way it is, and why it was chosen over other design alternatives.

3. System Overview

It can be said that monolithic applications are slowly becoming a thing of the past, with microservices rapidly taking their place, as the current state-of-the-art when it comes to system design. With many famous companies, such as Netflix or Amazon, following their approach, microservices undoubtedly have a massive role to play in software engineering of this age. Building the new InfluenzaNet system means building it for the future, and that requires following state-of-the art approaches and technologies – such as microservices.

To put simply, microservices architecture decomposes a large, complex system into loosely coupled services based on the functionality they provide. These services can then be implemented, tested, deployed, maintained, monitored and scaled completely independently from other services.

The picture below shows an informal view of the InfluenzaNet system and its services. It also shows how the communication flows between different components of the system. Instead of one large database, there are many smaller ones, each private to its own service. This was done in order to ensure loose coupling between the services.

Note that the application doesn't necessarily need to have separate user interfaces for each role. Whether the user interfaces will be separate, unique, or separate with a single entry point, shall yet to be determined.

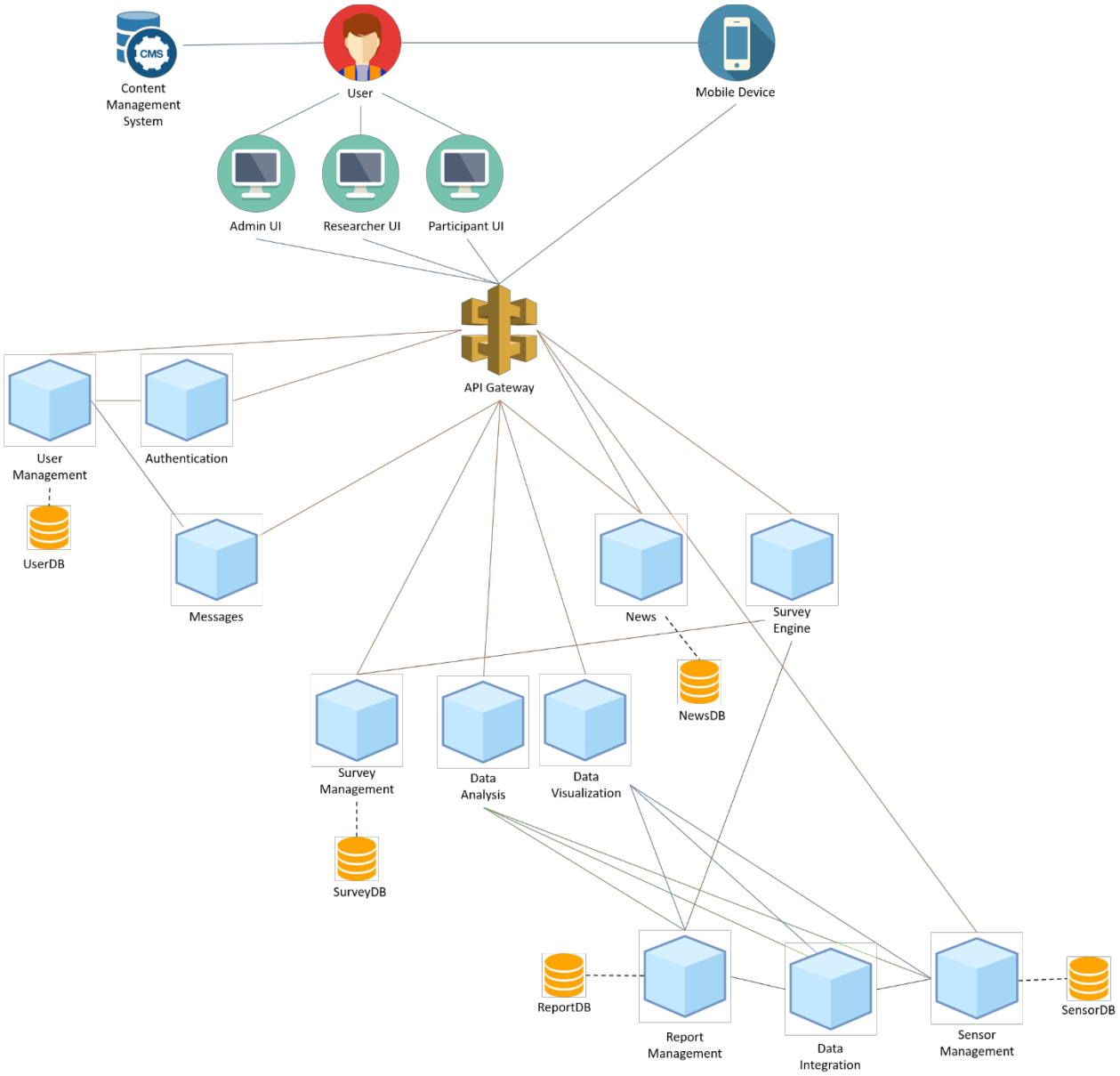

Figure 1. System overview

This section provided just an overview of the general idea behind the system architecture. For more details, please refer to Section 4. Additionally, Section 6 provides the rationale behind this architecture decision.

4. The Views

The following sub-chapters explain the four architectural views in detail. For a detailed description on how a view is documented, see Section 2.

4.1 Logical View

4.1.1 Primary Presentation

Logical view shows the static architecture, or in other words: it shows classes and relationships between them in a static environment (when they are not executing). This view may be useful to various types of stakeholders, as it shows what the system is supposed to do, and how.

Below is a class diagram drawn using UML notation.

Note that in this context “class” means a group of software components that may or may not make up an object-oriented class. They correspond more to a name “service” then “class”. The goal of this view is to show the minimum functionalities that each service shall provide, as well as indicate which services need to communicate to each other.

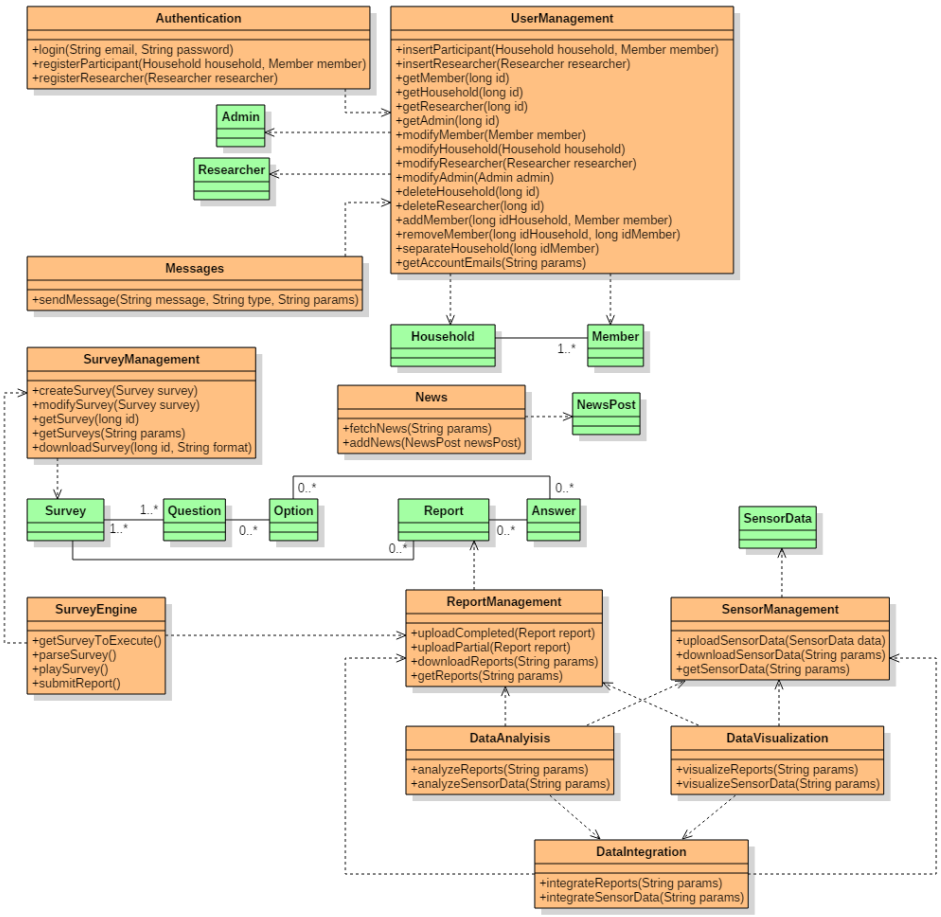

Figure 2. Class diagram

4.1.2 Element Catalog

4.1.2.1 Elements and their properties

Element	Properties
Authentication	A service that provides functionalities for authentication (log in, register).
UserManagement	A service that provides functionalities for managing user accounts.
SurveyManagement	A service that provides functionalities for managing surveys.
SurveyEngine	A service that parses and executes the surveys. This is a special service, as it exists both server-side and client-side.
ReportManagement	A service that provides functionalities for managing submitted reports.
SensorManagement	A service that provides functionalities for managing collected sensor data.
DataAnalysis	A service that provides a way to analyze collected results.
DataVisualization	A service that provides a way to visualize collected results.
DataIntegration	A service that integrates reports and sensor data from multiple countries.
Messages	A service that allows for sending out messages of different types to a specified set of accounts.
News	A service for providing ILI-related news.
Survey, Question, Option, Report, Answer, Household, Member, Researcher, SensorData, Admin, NewsPost	Correspond to database models explained in detail in the requirements document.

4.1.2.2 Relations and their properties

Relation	Between	Properties
Dependency	Authentication → UserManagement UserManagement → Admin UserManagement → Researcher UserManagement → Member UserManagement → Household Messages → UserManagement News → NewsPost SurveyManagement → Survey SurveyEngine → SurveyManagement SurveyEngine → ReportManagement ReportManagement → Report SensorManagement → SensorData DataAnalysis → ReportManagement DataAnalysis → SensorManagement DataVisualization → ReportManagement DataVisualization → SensorManagement	Shows that the class to the left of the arrow requires, needs or depends on the class to the right of the arrow.

	DataIntegration → ReportManagement DataIntegration → SensorManagement DataAnalysis → DataIntegration DataVisualization → DataIntegration	
Association	Survey – Question Question – Option Report – Answer Answer – Option Survey – Report Household - Member	Shows that the associated classes are connected (have a reference for one another).

4.1.2.3 Element interfaces

Element	Interfaces
Authentication	• *login* → attempts log in of a user given his e-mail and password • *registerParticipant, registerResearcher* → attempts registration of a new user if an account doesn't exist already
UserManagement	• *insertParticipant, insertResearcher* → inserts a new user to the database • *getMember, modifyMember*, → standard member management operations • *getHousehold, modifyHousehold, deleteHousehold* → standard household management operations • *getResearcher, modifyResearcher, deleteResearcher* → standard researcher management operations • *getAdmin, modifyAdmin* → standard admin management operations • *addMember* → adds new member to the household • *removeMember* → removes a member from the household • *separateHousehold* → separates a member into a new household and makes him an account holder • *getAccountEmails* → gets all account e-mails based on specified parameters
SurveyManagement	• *createSurvey* → creates a new survey • *modifySurvey* → modifies an existing survey • *getSurvey* → gets one survey • *getSurveys* → gets all surveys based on specified parameters • *downloadSurvey* → prepares survey for download

SurveyEngine	• *getSurveyToExecute* → gets the survey to execute based on the specified survey flow (invokes SurveyManagement service) • *parseSurvey* → parses a survey in a format required for its dynamic execution • *playSurvey* → executes the survey • *submitReport* → submits the report to the report database (invokes ReportManagement service)
ReportManagement	• *uploadCompleted* → uploads a completed report • *uploadPartial* → uploads a partially completed report • *downloadReports* → prepares reports for download • *getReports* → fetches all reports based on specified parameters
SensorManagement	• *uploadSensorData* → uploads sensor data • *downloadSensorData* → prepares sensor data for download • *getSensorData* → fetches all sensor data based on specified parameters
DataAnalysis	• *analyzeReports, analyzeSensorData* → support for data analysis
DataVisualization	• *visualizeReports, visualizeSensorData* → support for data visualization
DataIntegration	• *integrateReports, integrateSensorData* → performs integration of data by fetching it from multiple country-specific databases
Messages	• *sendMessage* → sends out a message of specified type; calls ParticipantManagement service to get a list of e-mails
News	• *fetchNews* → fetches ILI-related news from the Web • *addNews* → adds a new blog post or news
Survey, Question, Option, Report, Answer, Household, Member, Researcher, SensorData, Admin, NewsPost	Contain only getter and setter methods for all their attributes (not shown on the diagram).

4.1.2.4 Element behavior

The architecture pattern applied on this view is the *Microservices* pattern, already discussed in Section 3.

Design pattern (related to the *Microservices* pattern) applied in this case is *Decompose By Business Capability*. It explains the way functionality is decomposed into separate, loosely coupled services.

In order to successfully decompose the system, two object-oriented design guidelines are followed:

- *Single Responsibility Principle* → a class (or, in this case, a service) should only have one reason to change
- *Common Closure Principle* → classes that change for the same reason should be in the same package

In order to make the services as cohesive as possible and have them implement only a small set of related functions, we decomposed the system based on its business capabilities (something that a business does in order to generate value). They were identified by analyzing the organization's purpose, structure and business processes.

Due to each country possibly having its own databases and services for managing reports and sensor data, it was necessary to provide a Data Integration Service, that will integrate data from various sources and allow the researchers to analyze and visualize it in one place.

4.1.3 Rationale

Decomposing by Business Capability allowed us to obtain a stable architecture, since the business capabilities are relatively stable and almost never change.

The services are loosely-coupled, as independent as possible and highly cohesive. This provides a way to split development among autonomous teams, each responsible for one or more services. These teams now have the purpose to provide business value to the organization, rather than pure technical features.

The rationale behind the use of *Microservices* architecture pattern can be found in Section 6.

4.2 Process View

4.2.1 Primary Presentation

Process view explains how various parts of the system communicate during execution. It shows the message flow between different processes in the system, which is of most use to testers and integrators.

Below is a sequence diagram drawn using UML notation.

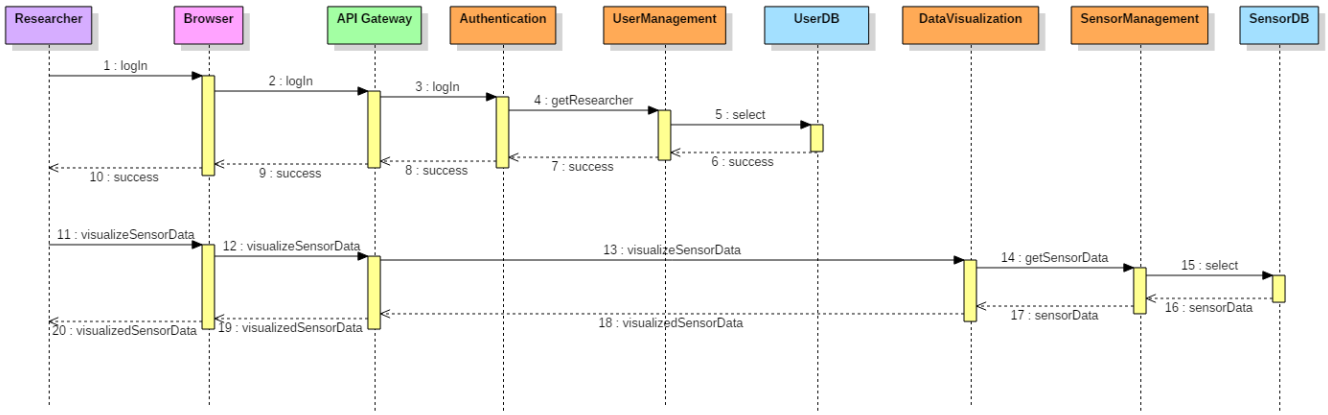

Figure 3. Sequence diagram

4.2.2 Element Catalog

4.2.2.1 Elements and their properties

Element	Type	Properties
Researcher	Role	A researcher that uses the system.
Browser	Browser	A Web browser that the researcher uses to access the system.
API Gateway	Object	A single entry point to the system.
Authentication	Object	An instance of Authentication class.
UserManagement	Object	An instance of UserManagement class.
UserDB	Database	Database containing user-related data.
DataVisualization	Object	An instance of DataVisualization class.
SensorManagement	Object	An instance of SensorManagement class.
SensorDB	Database	Database containing sensor-related data.

4.2.2.2 Relations and their properties

Relation	Properties
Message	Represents a message exchanged between the elements in order to successfully carry out the desired functionality of the system.
Reply	Represents a return value of a method.

4.2.2.3 Element interfaces

The interfaces of specific classes are explained in Section 4.1.2.3.

4.2.2.4 Element behavior

The interaction diagram above explains one specific use case: a researcher logs in to the system and wishes to see visualized sensor data.

The goal was to show how services communicate with each other to produce a desired result. Additionally, it illustrates the property of microservices that each service should have its own database. This particular design pattern is called *Database Per Service*.

The handling of other use cases is analogous to the one pictured above.

The services communicate with each other using well-defined service APIs. There are three types of communication between the services in this system:

1) Synchronous (a client makes a request and waits for a response)
2) Asynchronous (a client makes a request, but doesn't wait for a response; instead, it is notified when the response arrives)
3) Notification (a client makes a request and no reply is expected)

All communications that invoke writes to a database are asynchronous, as writes are expensive and can last some time. Notifications are used when sending messages, as the sender does not require any response. All other communications are synchronous.

Some services are exposed in order for external clients to access them. However, this is not done directly, but through the *API Gateway*, that represents a single point of entry to the system. It also implements the patterns *Circuit Breaker* and *Access Token*, which are crucial for correct functioning of the services.

It is important to ensure that a network failure happening on one service won't cascade to other services. For this we use the *Circuit Breaker* pattern. When a service cannot be reached a specified number of times, the *Circuit Breaker* ensures that for the duration of a timeout period all attempts to invoke that service will fail immediately. After the timeout period, a limited number of test requests is allowed to pass through. If they succeed, normal operation of the service is established. Otherwise, the timeout period starts again.

Finally, it is necessary to make sure that only authorized parties can access a specific service. This is done by having authenticated parties receive an access token that securely identifies them. A service can include this token in requests it makes to other services. This pattern is called *Access Token*.

4.2.3 Rationale

All requests from clients first go through the API Gateway (similar to the Façade object-oriented pattern). The API Gateway encapsulates the internal structure of the application, and is also responsible for request routing, composition and protocol translation. Using this approach offers many benefits, some of which are:

- It simplifies the communication between the clients and the services
- It can handle a request by invoking multiple microservices and aggregating the results

- It handles authorization and prevents cascading failures
- It can translate between web protocols (HTTP) and web-unfriendly protocols that are used internally
- It can offer different API for different devices
- If a service API is modified, it only needs to be updated in the API Gateway, and not on all clients individually

In order to enable loose coupling of services, as well as their independent development, deployment and scaling, it was necessary to decouple the database as well. Effectively, the database of the service is an integral part of that service's implementation, and its data can only be accessible via API provided by the service.

Additionally, this allows the developers to use different types of databases for different services, depending on their needs.

If data from multiple services is requested, an *API Composition* approach is applied, meaning that the application is the one doing the join rather than the database.

4.3 Physical View

4.3.1 Primary Presentation

Physical View shows where the packages fit on various physical parts of the system. It models the physical deployment of artifacts on nodes. This view is mostly useful to network administrators, and anyone working on communication between different parts of the system.

Below is a deployment diagram drawn using UML notation.

Note that the diagram is very generic; specific services and databases are not shown, and only terms such as "service" and "database" are used. The reason for this is the fact that the actual placement of containers on different machines is unknown and should not be influenced by the developer in any way.

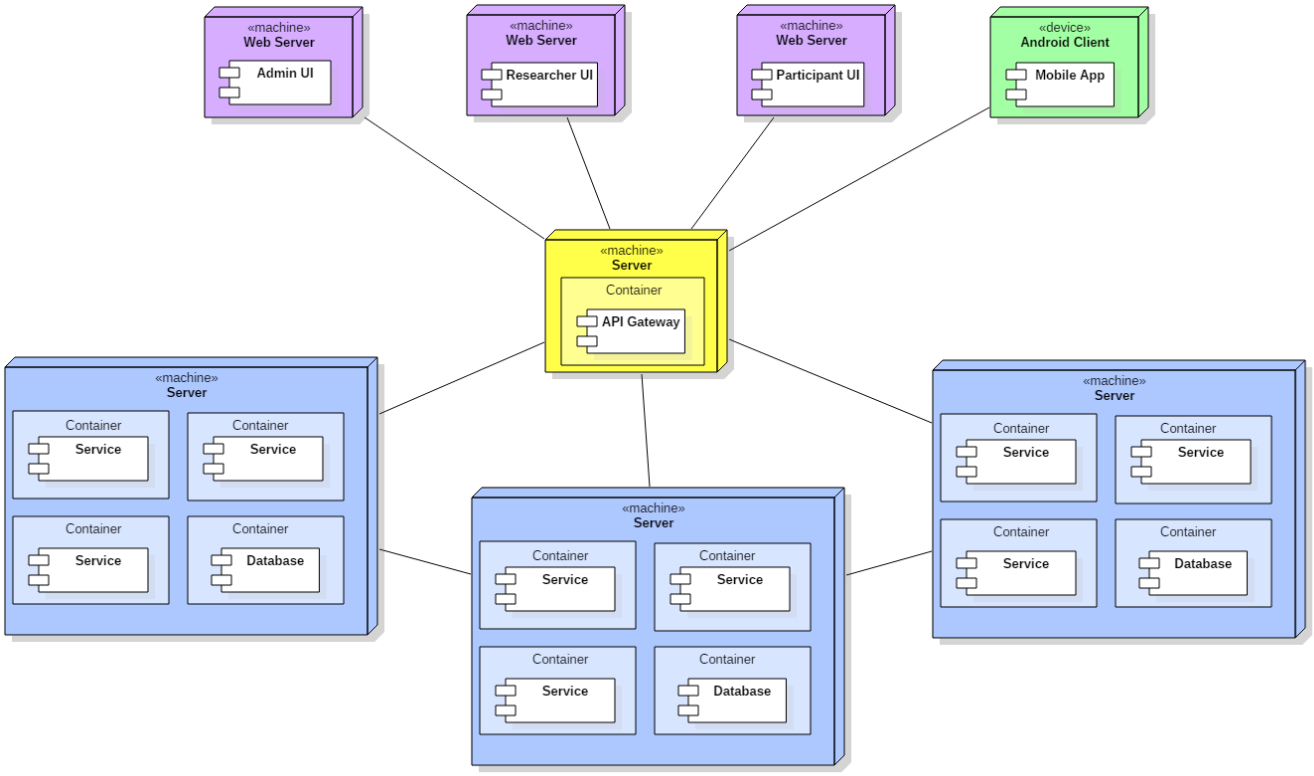

Figure 4. Deployment diagram

4.3.2 Element Catalog

4.3.2.1 Elements and their properties

Unlike the deployment diagram, the table below shows specific services.

Element	Type	Properties
Web Server	Node	A client server on which a Web application is run.
Android Client	Node	A client device on which a mobile application is run.
Container	Artifact	A lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it.
Admin UI	Artifact	A Web application for administrators that uses the APIs provided by the services to handle user requests.
Researcher UI	Artifact	A Web application for researchers that uses the APIs provided by the services to handle user requests.
Participant UI	Artifact	A Web application for participants that uses the APIs

		provided by the services to handle user requests.
Mobile Application	Artifact	A mobile application that uses the API provided by the backend application to handle user requests.
Server	Node	A server on which the services are run.
API Gateway	Artifact	A component implemented by the classes belonging to the “gateway” package.
Authentication Service	Artifact (Service)	A component implemented by the classes belonging to the “authentication” package.
User Management Service	Artifact (Service)	A component implemented by the classes belonging to the “users” package.
Survey Management Service	Artifact (Service)	A component implemented by the classes belonging to the “surveys” package.
Report Management Service	Artifact (Service)	A component implemented by the classes belonging to the “reports” package.
Data Analysis Service	Artifact (Service)	A component implemented by the classes belonging to the “analysis” package.
Data Visualization Service	Artifact (Service)	A component implemented by the classes belonging to the “visualization” package.
Data Integration Service	Artifact (Service)	A component implemented by the classes belonging to the “integration” package.
Survey Engine Service	Artifact (Service)	A component implemented by the classes belonging to the “engine” package.
Sensor Management Service	Artifact (Service)	A component implemented by the classes belonging to the “sensors” package.
News Service	Artifact (Service)	A component implemented by the classes belonging to the “news” package.
Messages Service	Artifact (Service)	A component implemented by the classes belonging to the “messages” package.
UserDB	Artifact (Database)	A database containing user-related data.
SurveyDB	Artifact (Database)	A database containing survey-related data.
ReportDB	Artifact (Database)	A database containing report-related data.
SensorDB	Artifact (Database)	A database containing sensor-related data.

NewsDB	Artifact (Database)	A database containing news-related data.

4.3.2.2 Relations and their properties

Relation	Between	Properties
Association	Web Server ↔ Server (through Internet) Android Client ↔ Server (through Internet) Server ↔ Server (through service APIs)	Shows that nodes on both sides of the relation are connected to each other.

4.3.2.3 Element interfaces

Element	Interfaces
Authentication Service	Has interfaces for accessing the realized classes belonging to the “authentication” package.
User Management Service	Has interfaces for accessing the realized classes belonging to the “users” package.
Survey Management Service	Has interfaces for accessing the realized classes belonging to the “surveys” package.
Report Management Service	Has interfaces for accessing the realized classes belonging to the “reports” package.
Data Analysis Service	Has interfaces for accessing the realized classes belonging to the “analysis” package.
Data Visualization Service	Has interfaces for accessing the realized classes belonging to the “visualization” package.
Data Integration Service	Has interfaces for accessing the realized classes belonging to the “integration” package.
Survey Engine Service	Has interfaces for accessing the realized classes belonging to the “engine” package.
Sensor Management Service	Has interfaces for accessing the realized classes belonging to the “sensors” package.
News Service	Has interfaces for accessing the realized classes belonging to the “news” package.
Messages Service	Has interfaces for accessing the realized classes belonging to the “messages” package.

4.3.2.4 Element behavior

There are two types of elements on the deployment diagram: nodes and artifacts. Nodes represent hardware components, while artifacts represent software components that run on nodes.

The Web Server/Android Client and the Servers containing the services are connected via Internet. The Web Server receives requests from users, serves requests for static content on its own, and contacts the services only in case dynamic content is needed.

The pattern used for deployment is *Service Instance Per Container*. This means that each service runs in its own container. The host usually contains multiple containers. It is also possible to have multiple instances of the same service, running either on the same host or on different ones. This introduces redundancy and avoids having a single point of failure.

4.3.3 Rationale

Using containers for deployment encapsulates the technology details used to build the service. In this way, all service instances are isolated from each other, and the amount of resources they can consume are limited by the container. Furthermore, the deployment is much faster and containers have a significantly shorter startup time than virtual machines (VMs).

Then, in a configuration like this, scalability is not an issue. If a specific service cannot handle the request load, it can easily be replicated as many times as necessary. A load balancer can also be added to ensure that all instances of a service receive an equal amount of requests.

Finally, there are many container clustering frameworks that can make the entire deployment straightforward and simple. Thanks to these frameworks, the person responsible for deployment does not have to worry about the actual placement of containers. The cluster manager makes the decision on where to place each container based on the container's resource requirements and available resources on each host.

4.4 Development View

4.4.1 Primary Presentation

Development view is of most use to developers and all other users that have a part in developing the system. It shows how different parts of the system are related and dependent on one another.

Below is a package diagram drawn using UML notation.

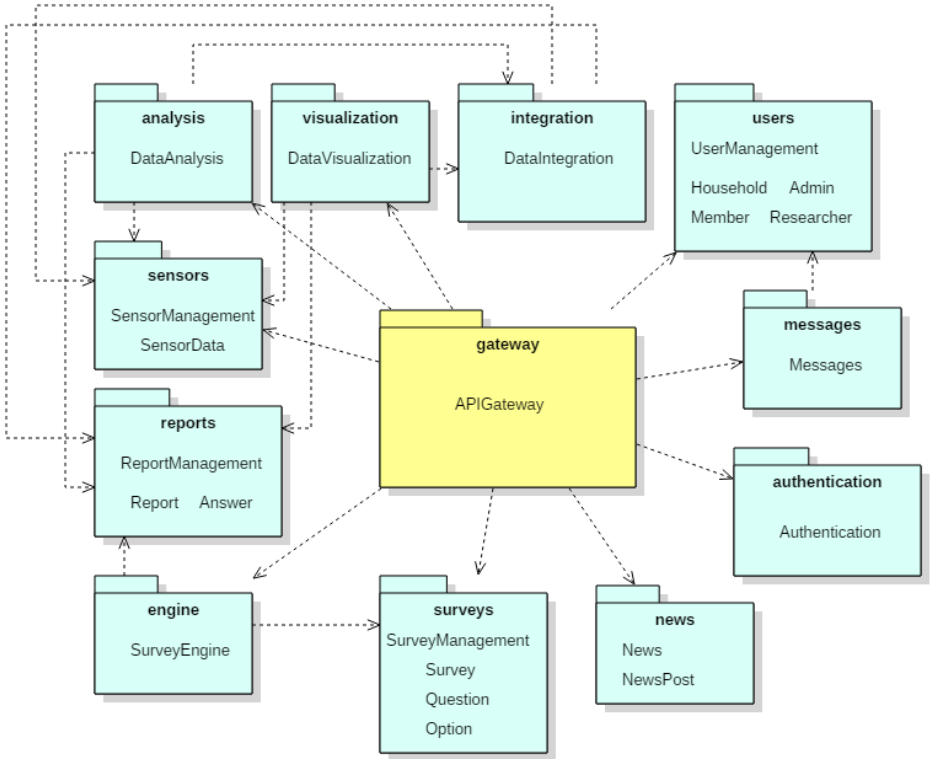

Figure 5. Package diagram

4.4.2 Element Catalog

4.4.2.1 Elements and their properties

The main elements of this view are packages. The view also contains classes, but their only purpose is to show which class belongs to which package. That is why classes are not listed in this section (they are described in detail in Section 4.1.).

Element	Properties
gateway	Groups together all classes required for accessing services.
users	Groups together all classes required for managing users.
authentication	Groups together all classes required for authentication.
surveys	Groups together all classes required for managing surveys.
reports	Groups together all classes required for managing submitted reports.
analysis	Groups together all classes required for analyzing collected data.
visualization	Groups together all classes required for visualizing collected data.
integration	Groups together all classes required for integrating data.
engine	Groups together all classes required for parsing and executing a survey.
sensors	Groups together all classes required for managing sensor data.
news	Groups together all classes required for managing news.
messages	Groups together all classes required for sending out messages.

4.4.2.2 Relations and their properties

Relation	Between	Properties
Dependency	gateway → users gateway → authentication gateway → news gateway → surveys gateway → engine engine → surveys engine → reports gateway → sensors gateway → analysis analysis → reports	Shows that at least one element of the package to the left of the arrow is dependent on an element from the package to the right of the arrow.

	analysis → sensors analysis → integration gateway → visualization visualization → reports visualization → sensors visualization → integration gateway → visualization integration → reports integration → sensors gateway → messages messages → users	

4.4.2.3 Element interfaces

The packages do not have any interfaces, while classes belonging to the packages do (as explained in Section 4.1.).

4.4.2.4 Element behavior

Each service is organized into a separate package based on its business capability, as mentioned in Section 4.1.2.4.

The gateway package is dependent on those services that the API Gateway requires in order to route requests.

The diagram above should indicate the order in which services shall be implemented. Independent services and those that do not require functionalities of other services shall be implemented first.

4.4.3 Rationale

In this view we can see the true benefits of the Microservices architecture; the fact that each service is organized into a separate package, as loosely coupled as possible with other packages, allows for easier separation of work among developers. Different services can be implemented by different development teams (even simultaneously).

The reasons behind decoupling services based on their business capability is explained in Section 4.1.3.

5. Mapping Between Views

The following tables explain how the views are connected to each other, and how they construct the architecture as whole.

Legend
LV – Logical View ProcV – Process View PhysV – Physical View

5.1 Logical View

Element	Is implemented by	Implements	Included in	Includes	Is an object of
User Management		User Management Service (PhysV)	users (DV)		
Member		User Management Service (PhysV)	users (DV)		
Household		User Management Service (PhysV)	users (DV)		
Admin		User Management Service (PhysV)	users (DV)		
Researcher		User Management Service (PhysV)	users (DV)		
Authentication		Authentication Service (PhysV)	authentication (DV)		
Survey Management		Survey Management Service (PhysV)	surveys (DV)		
Survey		Survey Management Service (PhysV)	surveys (DV)		
Question		Survey Management Service (PhysV)	surveys (DV)		
Option		Survey Management Service (PhysV)	surveys (DV)		
Report Management		Report Management Service (PhysV)	reports (DV)		
Report		Report Management Service (PhysV)	reports (DV)		
Answer		Report Management Service (PhysV)	reports (DV)		
Sensor Management		Sensor Management Service (PhysV)	sensors (DV)		
Sensor		Sensor Management Service (PhysV)	sensors (DV)		
SurveyEngine		Survey Engine Service (PhysV)	engine (DV)		
DataAnalysis		Data Analysis Service (PhysV)	analysis (DV)		

Data Visualization		Data Visualization Service (PhysV)	visualization (DV)		
DataIntegration		Data Integration Service (PhysV)	integration (DV)		
Messages		Messages Service (PhysV)	messages (DV)		
News		News Service (PhysV)	news (DV)		
NewsPost		News Service (PhysV)	news (DV)		

5.2 Process View

Element	Is implemented by	Implements	Included in	Includes	Is an object of
Researcher					
Browser					
API Gateway					
Authentication					Authentication (LV)
User Management					User Management (LV)
UserDB					
DataVisualization					DataVisualization (LV)
SensorManagement					Sensor Management (LV)
SensorDB					

5.3 Physical View

Element	Is implemented by	Implements	Included in	Includes	Is an object of
Web Server					
Android Client					
Container					
Admin UI					
Researcher UI					
Participant UI					
Mobile Application					
Server					
API Gateway					

Authentication Service	Authentication (LV)				
User Management Service	UserManagement, Member, Household, Admin, Researcher (LV)				
Survey Management Service	SurveyManagement, Survey, Question, Option (LV)				
Report Management Service	ReportManagement, Report, Answer (LV)				
Data Analysis Service	DataAnalysis (LV)				
Data Visualization Service	DataVisualization (LV)				
Data Visualization Service	DataIntegration (LV)				
Survey Engine Service	SurveyEngine (LV)				
Sensor Management Service	SensorManagement (LV)				
News Service	News (LV)				
Messages Service	Messages (LV)				
UserDB					
SurveyDB					
ReportDB					
SensorDB					
NewsDB					

5.4 Development View

Element	Is implemented by	Implements	Included in	Includes	Is an object of
gateway				API Gateway	
authentication				Authentication (LV)	
users				UserManagement, Member, Household, Admin, Researcher (LV)	
surveys				SurveyManagement, Survey, Question, Option (LV)	
engine				SurveyEngine (LV)	
reports				ReportManagement, Report, Answer (LV)	
analysis				DataAnalysis (LV)	

visualization				DataVisualization (LV)	
integration				DataIntegration (LV)	
sensors				SensorManagement (LV)	
news				News, NewsPost (LV)	
messages				Messages (LV)	

6. Rationale

Monolithic applications that put all functionality into a single process, despite their many advantages unfortunately face many problems when it comes to large, complex applications. As they grow, they become increasingly difficult to understand, maintain and test. As a result, this also slows down the development by a large margin. However, it is not only the development that suffers – the fact that the entire application needs to be redeployed every time a change is made, makes the continuous deployment virtually impossible.

It can be said that monolithic applications are slowly becoming a thing of the past, with microservices rapidly taking their place, as the current state-of-the-art when it comes to system design. They go hand in hand with many important advancements in software engineering, such as Agile software development, cloud computing, DevOps culture, continuous integration and continuous deployment, as well as the use of containers to package and deploy applications.

Even though dividing the application into services doesn't change the overall complexity of the problem, it makes it easier to grasp by breaking it up into smaller, manageable pieces. Additionally, the fact that any changes to a service impact only that service allows for easier maintenance and deployment.

Decoupling functionalities into separate services allows us to divide work among development teams more easily. And not only that – these teams can also be very diverse in terms of technologies they use. There are no limitations to combining services implemented using different technologies.

Another notable difference between a monolithic and microservices application is in a way it scales. While a monolithic application scales by replicating the monolith on multiple servers, microservices architecture enables the services to scale independently from each other, by replicating a specific service as many times as necessary.

Even though microservices also carry a certain number of disadvantages (such as added complexity of implementing and maintaining a distributed system), they are certainly the way to go for complex, evolving applications such as this one. Their many benefits will become even more apparent in the future, as the application continues to grow.

7. Glossary

Term	Meaning
ILI	Influenza-Like Illness
System	InfluenzaNet Backend Application
VM	Virtual Machine