



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION**

BSc THESIS

**A Web-Based Survey Manager for Building Dynamic Surveys
with Nested Visualizations**

Evangelos A. Garaganis

SUPERVISORS: **Yannis Smaragdakis**
Professor at National & Kapodistrian University of Athens

Kostas Saidis
Visiting Lecturer at National & Kapodistrian University of Athens

ATHENS

11/2020



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

**A Web-Based Survey Manager for Building Dynamic Surveys
with Nested Visualizations**

Ευάγγελος Α. Γκαραγκάνης

Επιβλέποντες: **Γιάννης Σμαραγδάκης**
Καθηγητής στο Καποδιστριακό Πανεπιστήμιο Αθηνών
Κώστας Σαΐδης
Λέκτορας στο Καποδιστριακό Πανεπιστήμιο Αθηνών

ΑΘΗΝΑ

11/2020

BSc THESIS

A Web-Based Survey Manager for Building Dynamic Surveys with Nested Visualizations

Evangelos A. Garaganis

S.N.: 1115201400033

SUPERVISORS: Yannis Smaragdakis

Professor at National & Kapodistrian University of Athens

Kostas Saidis

Visiting Lecturer at National & Kapodistrian University of Athens

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

A Web-Based Survey Manager for Building Dynamic Surveys with Nested Visualizations

Ευάγγελος Α. Γκαραγκάνης

A.M.: 1115201400033

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης**
Καθηγητής στο Καποδιστριακό Πανεπιστήμιο Αθηνών

Κώστας Σαΐδης
Λέκτορας στο Καποδιστριακό Πανεπιστήμιο Αθηνών

ABSTRACT

During this thesis, we combined UI/UX disciplines, WWW technologies and data analytics techniques and tools, to create a web-based survey manager SaaS (Software as a service) for building dynamic surveys with nested visualizations.

With our service, users will be able:

- To create different types of questions.
- Build surveys with the questions created.
- Visualize the surveys' answers to the questions.
- Experiment with the results by combining a number of questions.
- Share surveys and results with the public.

The platform was created with the usage of cutting-edge technologies and reached the full-stack spectrum of developing. The whole implementation is written purely on JS, by using the frameworks React and Material UI for the front-end and Express NodeJs' framework for the back-end. The data-storing and data-analyzing is being handled by the Elasticsearch engine, while the data-visualization is made by the ReCharts charting library. Finally, we used the RESTful architecture for the data exchange and NPM for the package management of the application.

SUBJECT AREA: Web Development

KEYWORDS: Data Analytics, Data Visualization, React, NodeJS, Elasticsearch,

ΠΕΡΙΛΗΨΗ

Κατά την διάρκεια αυτής της πτυχιακής, συνδυάσαμε αρχές ευχρηστίας, τεχνολογίες του παγκόσμιου ιστού και εργαλεία και τεχνικές ανάλυσης και οπτικοποίησης δεδομένων, για να δημιουργήσουμε μια πλατφόρμα δημιουργίας ερωτηματολογίων και ανάλυσης αποτελεσμάτων. Με αυτή την πλατφόρμα οι χρήστες θα μπορούν:

- Να δημιουργούν ερωτήσεις διαφόρων τύπων.
- Να κατασκευάζουν ερωτηματολόγια με τις ερωτήσεις που έχουν δημιουργήσει.
- Να οπτικοποιήσουν τις απαντήσεις του ερωτηματολογίου.
- Να πειραματιστούν με τα αποτελέσματα, συνδυάζοντας τις απαντήσεις πολλών ερωτήσεων.
- Να δημοσιεύσουν τα ερωτηματολόγια τους και τα αποτελέσματα αυτών στο ευρύ κοινό.

Το λογισμικό αυτό δημιουργήθηκε με την χρήση κορυφαίων τεχνολογιών που χρησιμοποιούνται σε όλο το φάσμα ανάπτυξης λογισμικού για τον παγκόσμιο ιστό. Ολόκληρη η υλοποίηση έχει γραφτεί σε Javascript, χρησιμοποιώντας React και Material UI για τον front-end και Express για το back-end. Η αποθήκευση και ανάλυση των δεδομένων γίνεται με την μηχανή αναζήτησης του Elasticsearch, ενώ η οπτικοποίηση των δεδομένων γίνεται με την βιβλιοθήκη Recharts. Τέλος, η εφαρμογή έχει δομηθεί με RESTful αρχιτεκτονική για την ανταλλαγή των δεδομένων και το NPM για την διαχείριση των packages που χρησιμοποιούνται.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάπτυξη Εφαρμογής Διαδικτύου

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανάλυση Δεδομένων, Οπτικοποίηση Δεδομένων, React,
NodeJS, Elasticsearch

ACKNOWLEDGEMENTS

For this thesis completion, I would like to deeply thank my instructor and supervisor Mr. Kostas Saidis for his guidance during my academic years . He is an example of a scientist and professional that I will live by. I also want to thank Mr. Yannis Smaragdakis and lecturers like him that inspired us to follow and commit to the path of Computer Science.

Last but not least, I am grateful to my family for giving me the opportunity to attend a university and for standing by my side throughout this journey.

ΕΥΧΑΡΙΣΤΙΕΣ

Για την εκπόνηση της παρούσας πτυχιακής εργασίας, θα ήθελα να ευχαριστήσω τον καθηγητή μου και επιβλέπων αυτής της πτυχιακής, τον κ. Σαΐδη, οποίος υπήρξε οδηγός κατά την διάρκεια των σπουδών μου και ο οποίος θα αποτελεί πρότυπο επιστήμονα και επαγγελματία για το μέλλον μου. Επίσης, θέλω να ευχαριστήσω τον κ.Σμαραγδάκη και καθηγητές σαν αυτόν, που μας ενέπνευσαν να ακολουθήσουμε τον δρόμο της επιστήμης της πληροφορικής.

Τέλος και κυριότερο, θα ήθελα να ευχαριστήσω την οικογένεια μου, που μου έδωσε την δυνατότητα να σπουδάσω και με στήριξε σε όλη αυτή την διαδρομή.

Στον παππού μου

To my grandfather

CONTENTS

ABSTRACT	5
ACKNOWLEDGEMENTS	7
CONTENTS	11
LIST OF FIGURES	14
LIST OF CODE EXAMPLES	17
LIST OF DIAGRAMS	19
LIST OF TABLES	21
PREFACE	22
1. INTRODUCTION	23
2. SYSTEM OVERVIEW	24
2.1 Question Creation	24
2.2 Questions Management	25
2.3 Survey Creation	25
2.4 Survey Form	27
2.5 Surveys management	28
2.6 Results Reporting	28
2.7 Visualization Wizard	30
2.8 Use Cases	31
3. SYSTEM DESIGN	32
3.1 Questions	32
3.1.1 Design	32
3.1.2 Modeling	34
3.1.3 Rendering	37
3.2 Surveys	42
3.2.1 Design	42
3.2.1.1 Storage Format	42
3.2.1.2 Runtime Format	43
3.2.1.3 Difference between the two survey formats	44
3.2.1.4 Conversion between the two formats and outline	45
3.2.2 Modeling	46
3.2.3 Rendering	48
3.3 Answered Surveys	52
3.3.1 Design	52

3.3.2 Exportation and Storage	54
3.4 Results Visualization	57
3.4.1 General	57
3.4.2 Results Reporting	58
3.4.2.1 Results Rendering	59
A. Terms Questions	60
B. Stats Questions	62
3.4.2.2 Statistics Provider	64
3.4.2.3 Summing up	66
3.4.3 Visualization Wizard	67
3.4.3.1 General	67
3.4.3.2 Analysis	73
I. Statistics Provider	74
i) Questions Selected Data to Query process	75
ii) Aggregations Results to Wizard Statistics process	89
II. Visualization Wizard Renderer	97
i) Questions Selected Data Building	98
ii) Wizard Stats Visualization	107
III. Summing up	111
3.5 UX and UI decisions	112
3.5.1 Usability Evaluation	112
3.5.2 Theming	114
3.5.3 Accessibility	115
3.6 Design Challenges	115
3.6.1 Architectural Decisions	115
3.6.2 Design Principles	118
4. SYSTEM IMPLEMENTATION	119
4.1 The technology stack used	119
4.2 System Architecture	120
4.2.1 Front-End	120
4.2.1.1 React	120
4.2.1.2 Material UI	121
4.2.1.3 Recharts	121
4.2.1.4 React Router	122
4.2.1.5 Local Storage	123
4.2.1.6 Axios and Data Fetching	124
4.2.1.7 React Context	124
4.2.2 Back-End	126
4.2.2.1 Rest API & Endpoints	126
4.2.2.2 ExpressJS	128
i) Routes	129

ii) Controllers	130
4.2.2.3 Elasticsearch	131
I. Index Organization	131
II. Queries	131
III. Elasticsearch JS Client	132
IV. Kibana	134
V. How Aggregations Work	134
VI. Full-Text Search	139
4.3 Summing Up	141
4.4 Technical Challenges	142
5. CONCLUSION	143
TABLE OF TERMINOLOGY	144
ABBREVIATIONS	145
REFERENCES	146

LIST OF FIGURES

Figure 1: Question Creation	24
Figure 2: Question Pool	25
Figure 3: Survey Creation	26
Figure 4: Survey Form	27
Figure 5: Surveys Management	28
Figure 6: RR Overview	29
Figure 7: RR Stats Question	29
Figure 8: RR Terms Question	29
Figure 9: Visualization Wizard	30
Figure 10: Dev-Study Use Case	31
Figure 11: Question Format	33
Figure 12: Text Input Question	41
Figure 13: Slider Question	41
Figure 14: Radio Button Question	41
Figure 15: Checkbox Question	41
Figure 16: Storage Format Structure	42
Figure 17: Example of Storage Format	43
Figure 18: Run Time Format	44
Figure 19: Rendered Survey	50
Figure 20: 3 Document Types	52
Figure 21: Answered Survey Example	53
Figure 22: Term Questions Statistics Format	60
Figure 23: Rendered Terms Question Graph	61
Figure 24: Stats Question Data Format	62
Figure 25: Rendered Stats Question Graph	63

Figure 26: Visualization Wizard Example 1	68
Figure 27: Visualization Wizard Example 1 - Parameterized	69
Figure 28: Visualization Wizard Example 2	70
Figure 29: Visualization Wizard Example 3	71
Figure 30: Question Selected Data Format	76
Figure 31: DataStore Query Format - Example 1	79
Figure 32: DataStore Query Format - Example 2	80
Figure 33: Digested Questions Selected Data	83
Figure 34: Before and After Aggregations Composition	85
Figure 35: Query Building Components	86
Figure 36: DataStore Query Built	87
Figure 37: Aggregation Results	89
Figure 38: Wizard Statistics	92
Figure 39: Questions Selected Data Building	98
Figure 40: Questions Selected Rendering Cards	100
Figure 41: Questions Selected Cards	101
Figure 42: Questions Card Parameters	102
Figure 43: Terms Questions Form Parameters	105
Figure 44: Stats Questions Form Parameters	105
Figure 45: VW Another Example	108
Figure 46: Theming and Color Palettes	112
Figure 47: Error Prevention Options	112
Figure 48: Graphic Designs	113
Figure 49: Light And Dark Theme	114
Figure 50: Wireframes	115
Figure 51: Component Architecture Logic	116
Figure 52: Continue Survey	123

Figure 52: Kibana	134
Figure 53: Full-Text Search	140

LIST OF CODE EXAMPLES

Code 1: Question Creation Code	35
Code 2: Question Class Code	36
Code 3: Material Checkbox Form Component	39
Code 4: Question Renderer Code	40
Code 5: Survey Creation	47
Code 6: Survey Class	47
Code 7: Survey Rendering - Survey Creation	48
Code 8: Survey Rendering - Methods	48
Code 9: Input Changed Handler	49
Code 10: Survey Submission	55
Code 11: Export Survey Method	56
Code 12: Visualization Renderer	60
Code 13: Terms Question Graph	62
Code 14: Stats Question Graph	63
Code 15: Results Reporting Statistics Provider	65
Code 16: Arguments To Aggregations Transforming	84
Code 17: Aggregations Composition	86
Code 18: Data Store Query Building	87
Code 19: Wizard Statistics Building	93
Code 20: Form Aggregation Parameters Renderer	104
Code 21: Visualization Wizard Stats Visualizer	109
Code 22: React Router	122
Code 23: Continue Survey	123
Code 24: Axios Data Fetching	124
Code 25: Update Questions Aggregation Data Method	125

Code 26: Server Implementation	128
Code 27: Routes	129
Code 28: Controllers	130
Code 29: Elastic JS Client getQuestions Method	132
Code 30: Elastic JS Client Methods	133
Code 31: Full-Text Search	140

LIST OF DIAGRAMS

Diagram 1 : Question Modeling	34
Diagram 2: Question Rendering	37
Diagram 3: Question Rendering	38
Diagram 4: Conversion Between Survey Formats	45
Diagram 5: Survey Class Model	46
Diagram 6: Survey Rendering Abstract	51
Diagram 7: Survey Exportation and Storage	54
Diagram 8: RR - Server Client Interaction	58
Diagram 9: Visualization Renderer	59
Diagram 10: RR Statistics Provider	64
Diagram 11: RR Complete Process	66
Diagram 12: VW Process Abstract	73
Diagram 13: VW Statistics Provider Abstract	74
Diagram 14: Data Transformation Process Abstract	75
Diagram 15: VW Statistics Provider - Phase 1	75
Diagram 16: Data Transformation Process - Phase 1	76
Diagram 17: VW Statistics Provider - Phase 2	77
Diagram 18: Data Transformation Process - Phase 2	78
Diagram 19: VW Statistics Provider - Phase 3	81
Diagram 20: Data Transformation Process - Phase 3	81
Diagram 21: Forward Transformation Process Abstract	82
Diagram 22: Data Transformation Process - Forward Complete	86
Diagram 23: Data Transformation Process - Phase 4	90
Diagram 24: VW Statistics Provider - Phase 4	90
Diagram 25: VW Statistics Provider - Phase 5	91

Diagram 26: Data Transformation Process - Phase 5	91
Diagram 27: Data Transformation Process - Backward Complete	94
Diagram 28: VW Statistics Provider - Phase 6	94
Diagram 29: Data Transformation Process - Phase 6	95
Diagram 30: VW Statistics Providing Complete	97
Diagram 31: VW Renderer Abstract	97
Diagram 32: BW Renderer Component Tree	99
Diagram 33: Passing Question Data	103
Diagram 35: VW Front-End Detailed	106
Diagram 36: VW Front-End Complete	110
Diagram 37: VW Complete	111
Diagram 38: Back-End Abstract Structure	117
Diagram 39: Back-End Structure Detailed	117
Diagram 40: Question Rendering Abstract	118
Diagram 41: Back-End Structure Detailed	119
Diagram 42: React Context	125
Diagram 43: Routes And Controllers	129
Diagram 44: Elasticsearch JS Client	132
Diagram 45: Documents In Index Without Aggregations	138
Diagram 46: Survey ID Match Aggregation	136
Diagram 47: Documents After Aggregation 0	137
Diagram 48: Documents After Aggregation 1	138
Diagram 49: Documents In Index Last Aggregation	139
Diagram 50: Full-Stack Abstract Architecture	141

LIST OF TABLES

Table 1: REST API Endpoints

125

PREFACE

This thesis was written within my undergraduate studies in the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, Greece, under the supervision and guidance of Prof. Yannis Smaragdakis and Dr. Kostas Saidis.

1. INTRODUCTION

In this thesis we created a web-based survey manager for building dynamic surveys with nested visualizations platform that helps users build and conduct simple, performant and customizable surveys. Individuals or corporations are given the option to create their own question pool, by selecting a broad range of question types and parameters, and then build their surveys based on those questions, by flexibly designing the sections and questions sequence. After building the surveys, users can publish them to the public, for a period specified, and let our platform gather the results. For each survey, the user has the option to get the report containing questions' answers, each one visualized with a different graph based on the question type. The platform also provides a way to experiment with the survey results, by allowing users to combine question answers with the help of our visualization wizard.

The major question considering our implementation is, why not use an already existing service for the survey building and reporting? The answer resides in the initial goal of this thesis. We wanted an all-in-one platform to handle our series of surveys, starting from the survey creation and concluding to the results visualization. Our top priority was an automated and flexible way to create different kinds of surveys that shared the same questions and a powerful method to visualize and experiment with the results. In that manner, we have the ability to analyze the results and extract our conclusions or grant the public access to the answers of those surveys, so that individuals can extract their own. The above all-in-one solution, given the fact that it was built with the help of cutting-edge technologies and the suitable architectural and design patterns, leads us to create a usable and performant survey building and analyzing platform that fits our needs end-to-end and enables us to add our own functionality, however we wish to. And of course, people that like our approach can do the same to fulfil their own needs.

2. SYSTEM OVERVIEW

2.1 Question Creation

I will start the tool showcase with the building stone of the surveys, the questions. Each question that can be used in a survey, has different types considering the question's tenor, and a set of rules that users can apply, to restrict the answers given. The user decides the question text and type, while also providing the essential information for the question. The platform live-renders the question results, helping him reach the desirable result. There is also some metadata about the question, like the estimated completion time or the surveys that the question exists in. Let's see some screenshots from our platform to understand the question management process.

Edit Question
Build, Edit & Preview your questions

Build the question

ID: Q01 Type: SetOfStrings

Text: Γιατί σκοπεύετε να ασχοληθείτε επαγγελματικά με την ανάπτυξη λογισμικού;

Hint: Μπορείτε να επιλέξετε παραπάνω από μια επιλογές

Values:

- Οικονομικά Κίνητρα
- Εξωτερικές Πιέσεις
- Εσωτερικά Κίνητρα
- Δεν είχα άλλες επιλογές
- Add a value

Rules:

Pick a rule

Preview result

1. Γιατί σκοπεύετε να ασχοληθείτε επαγγελματικά με την ανάπτυξη λογισμικού;

- ☐ Οικονομικά Κίνητρα
- ☐ Εξωτερικές Πιέσεις
- ☐ Εσωτερικά Κίνητρα
- ☐ Δεν είχα άλλες επιλογές

🕒 Estimated completion time: 30 sec

Question resides in the following:

STUD Survey

Figure 1: Question Creation

2.2 Questions Management

Για την διαχείριση των ερωτήσεων που έχει φτιάξει ο χρήστης, του δίνεται η δυνατότητα να πλοηγηθεί σε αυτές, τόσο μέσω της μηχανής αναζήτησης η οποία λειτουργεί για οποιοδήποτε πεδίο της ερώτησης (με την χρήση full-text search), όσο και με την λίστα των ερωτήσεων οι οποίες μπορούν να ταξινομηθούν βάσει διαφόρων πεδίων. Από αυτή την σελίδα ο χρήστης μπορεί να δημιουργεί ερωτήσεις, να επεξεργάζεται τις ήδη υπάρχων ή να τις διαγράφει.















Question Pool			
Manage & Organize your questions			
<input type="text" value="Search for a question type, text, other..."/>			+
Id	Type	Text	Actions
Q01	SetOfStrings	Γιατί σκοπεύετε να ασχοληθείτε επαγγελματικά με την ανάπτυξη λογισμικού;	 
Q02	String	Κρίνετε τον εαυτό σας επαρκώς προετοιμασμένο για τη μετάβαση από τις σπουδές στην αγορά εργασίας;	 
Q03	SetOfStrings	Ποιοι είναι οι μεγαλύτεροι φόβοι που νιώθετε κατά τη μετάβαση αυτή;	 
Q04	String	Θεωρείτε ότι οι σπουδές σας έχουν προσφέρει τα απαραίτητα εφόδια για τη μετάβαση αυτή;	 
Q05	SetOfStrings	Τι θα αλλάζατε σχετικά με τον τρόπο που αντιμετωπίσατε εσείς τις σπουδές σας και γιατί;	 
Q06	String	Είστε ικανοποιημένοι από την ποιότητα των σπουδών σας	 
Q07	String	Έχετε εργαστεί στο αντικείμενο στη διάρκεια των σπουδών σας;	 

Figure 2: Question Pool

2.3 Survey Creation

Now that the user has a range of questions to choose by, they can create a survey. Starting with the survey basic information, like the title, description, the time period that the survey will be accessible, and then deciding the survey sectioning and question selection, users can build their survey how they wish to.


The builder strives for usability, providing drag & drop capabilities to reorder and reindex the survey's sections and questions, while also giving the option to real-time render and preview the result of the final survey.

Finally, the survey builder also contains extra information for the survey, like the estimated completion time, that is calculated with our algorithms.

Questions: 35
Takes around: 17.5 min

Survey's Basic Info

Survey Icon



Survey's ID

STUD

Ends on

08/18/2014

Survey's Title

Ερωτηματολόγιο Φοιτητών

Short Description

Για φοιτητές έτοιμους να βγούν στην αγορά εργασίας

Detailed Description

Το ερωτηματολόγιο απευθύνεται σε τελειόφοιτους τμημάτων πληροφορικής, οι οποίοι ενδιαφέρονται να ασχοληθούν επαγγελματικά με την ανάπτυξη λογισμικού (Software Engineering and Development).

Section 1/ Σχολή και Μετάβαση στην αγορά εργασίας

Section 2/ Το well-being του προγραμματιστή

Section 3/ Βασικές Πληροφορίες

Title

Βασικές Πληροφορίες

Description

Βασικά στοιχεία σχετικά με τον ερωτηθέντα του ερωτηματολογίου

Questions In Section:

ID: Q33

Text:

Σε ποιά τμήμα πληροφορικής σπουδάζετε;

ID: Q34

Text:

Σε ποιά έτος φοίτησης βρίσκεστε;

ID: Q35

Text:

Τρέχον βαθμός πτυχίου;

ID: Q36

Text:

Το φύλλο σας;

Manage questions

Add section

PREVIEW RESULT

More Options:

Print

Publish

Restore

Delete

SAVE CHANGES

Cancel

Figure 3: Survey Creation

E. Garaganis

26

2.4 Survey Form

Whenever the user feels satisfied with the resulting survey or wants to preview its current state, they can click **Preview Result**, to inspect the resulting survey form. An example of the survey form can be:

Ερωτηματολόγιο Φοιτητών

Το ερωτηματολόγιο απευθύνεται σε τελειόφοιτους τμημάτων πληροφορικής, οι οποίοι ενδιαφέρονται να ασχοληθούν επαγγελματικά με την ανάπτυξη λογισμικού (Software Engineering and Development).

Το well-being του προγραμματιστή [2/3]

Οι ερωτήσεις στο παρακάτω κομμάτι έχουν να κάνουν με τις συνήθειες και την υγεία του προγραμματιστή

1. Βαθμολογήστε την ποιότητα των συνηθειών σας (διατροφή, άσκηση, ποιότητα ύπνου...).

0 1 2 3 4 5 6 7 8 9 10

2. Είστε ευχαριστημένοι από την κοινωνική σας ζωή;

0 1 2 3 4 5 6 7 8 9 10

3. Η επαγγελματική ενασχόληση κρίνετε πως θα τον επιβαρύνει;

☐ Ναι ☐ Όχι

4. Οι σπουδές σας πόσο τον άλλαξαν;

☐ Καθόλου ☐ Λίγο ☐ Αρκετά ☐ Πολύ

◀ Προηγούμενο
Επόμενο ▶

Figure 4: Survey Form

The rendered survey contains the questions that the builder decided, with the sectioning and order specified, while also having the Survey Helper on the bottom right corner of the screen that informs the user about questions containing errors or that haven't been answered yet, auto-navigating to them.

2.5 Surveys management

The surveys created are listed in the admin page of the platform. From this page, the user can navigate to edit the survey, to the page for the survey completion (if the user has decided to publish the survey) or the survey results for the published surveys that contain results. Each survey card contains indicators specifying if the survey or its results are live or not, and information of the survey date created etc. From the admin page it is also possible to go to the question management page or create a new survey.

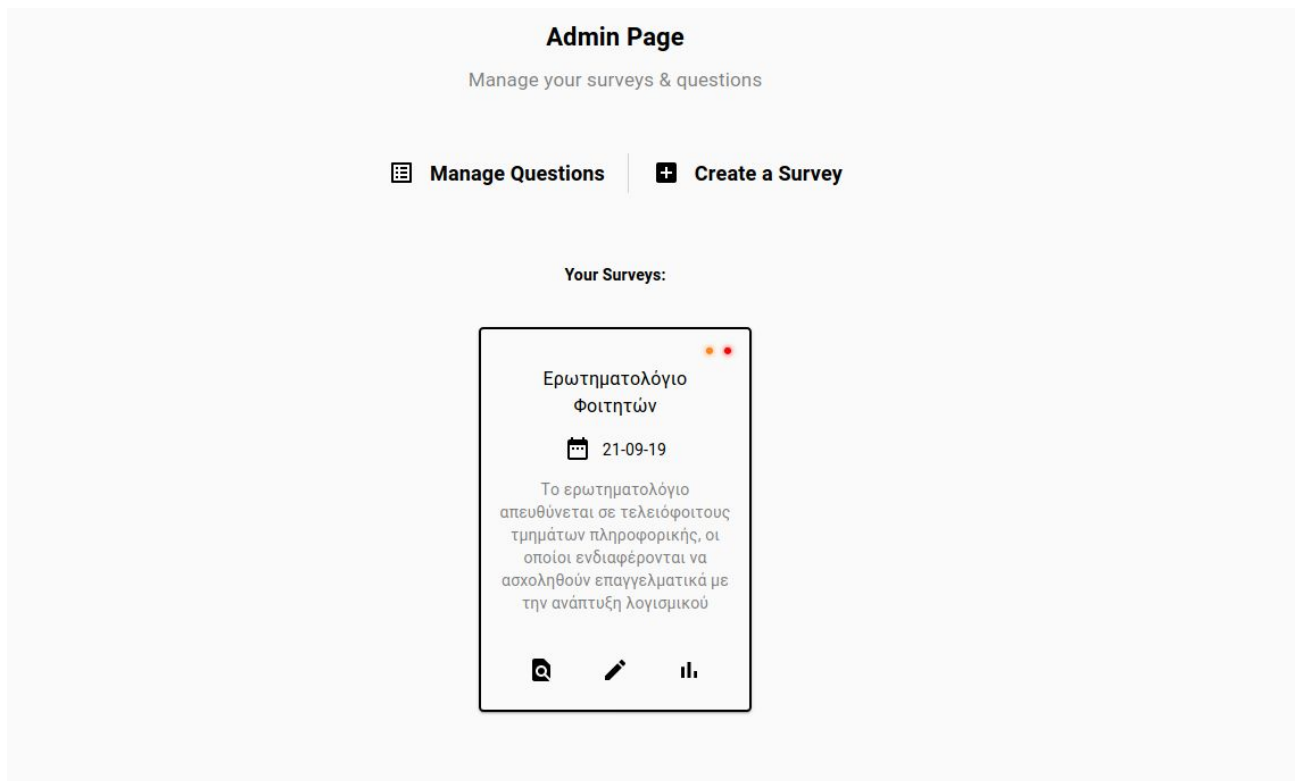


Figure 5: Surveys Management

2.6 Results Reporting

For the surveys that are published, the public can take and submit the surveys. Each completed survey is stored within our systems in the appropriate format and ready for our service to analyze it and produce the results. The **Results Reporting** page contains the table of contents with the survey's sections and questions, the question results given with the graph for each data type and the option to share the results or download the full report in a pdf form. Let us see some screens to visualize what we are saying.



Figure 6: Results Reporting Overview

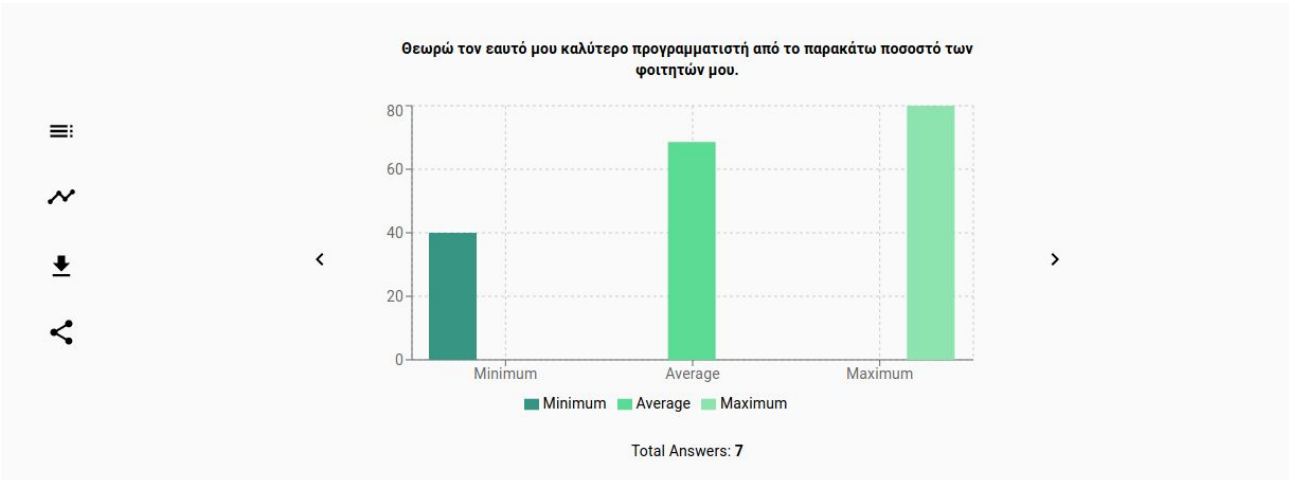


Figure 7: Results Reporting Stats Question

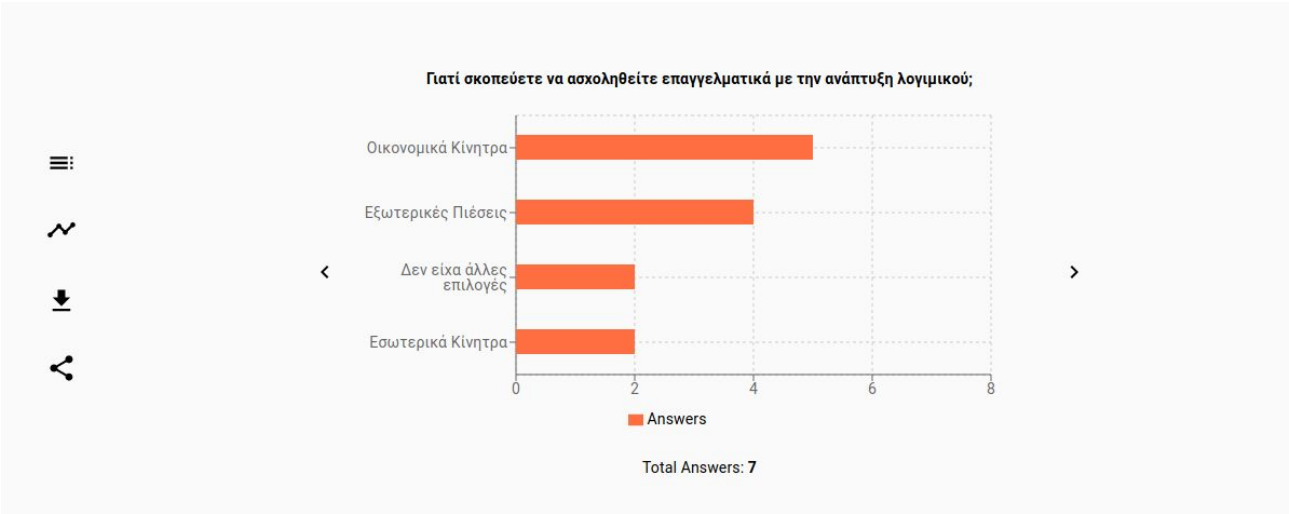


Figure 8: Results Reporting Terms Question

2.7 Visualization Wizard

Given the same answers for the surveys, users can navigate from the **Report Page** to **Visualization Wizard**. The Visualization Wizard is a powerful feature that allows individuals to parameterize the question answers based on their fields or ranges, combine an arbitrary number of questions and plot the resulting statistics in an appealing and easy to digest visual way.

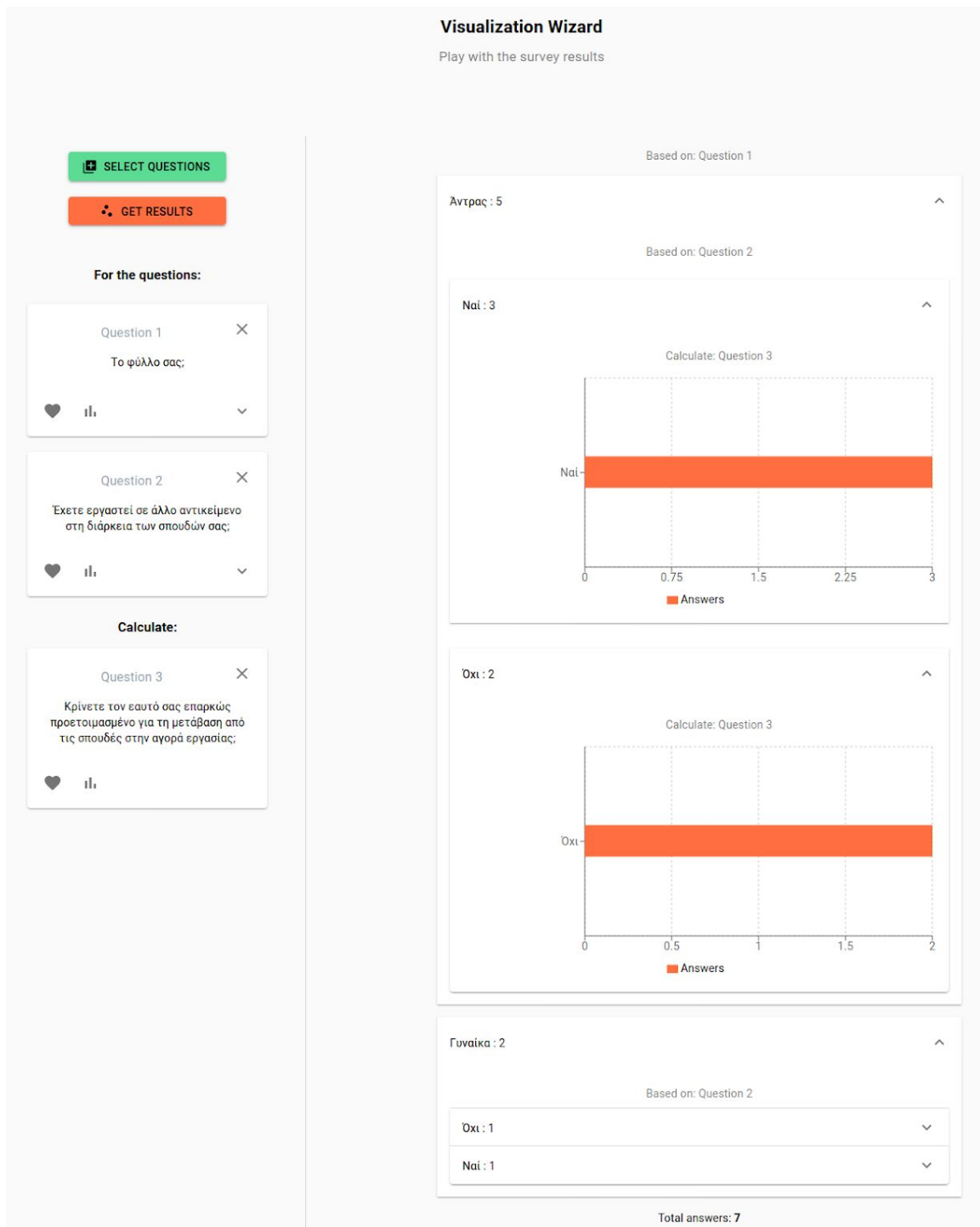


Figure 9: Visualization Wizard

2.8 Use Cases

The whole process of the survey building, publishing and results visualization and experimentation can happen for any kind of survey and can be integrated in any platform. We are using our service to conduct our series of surveys that concerns only developers and the computer science spectrum. We will showcase the use case, but any corporation can integrate our solution for their own purposes.

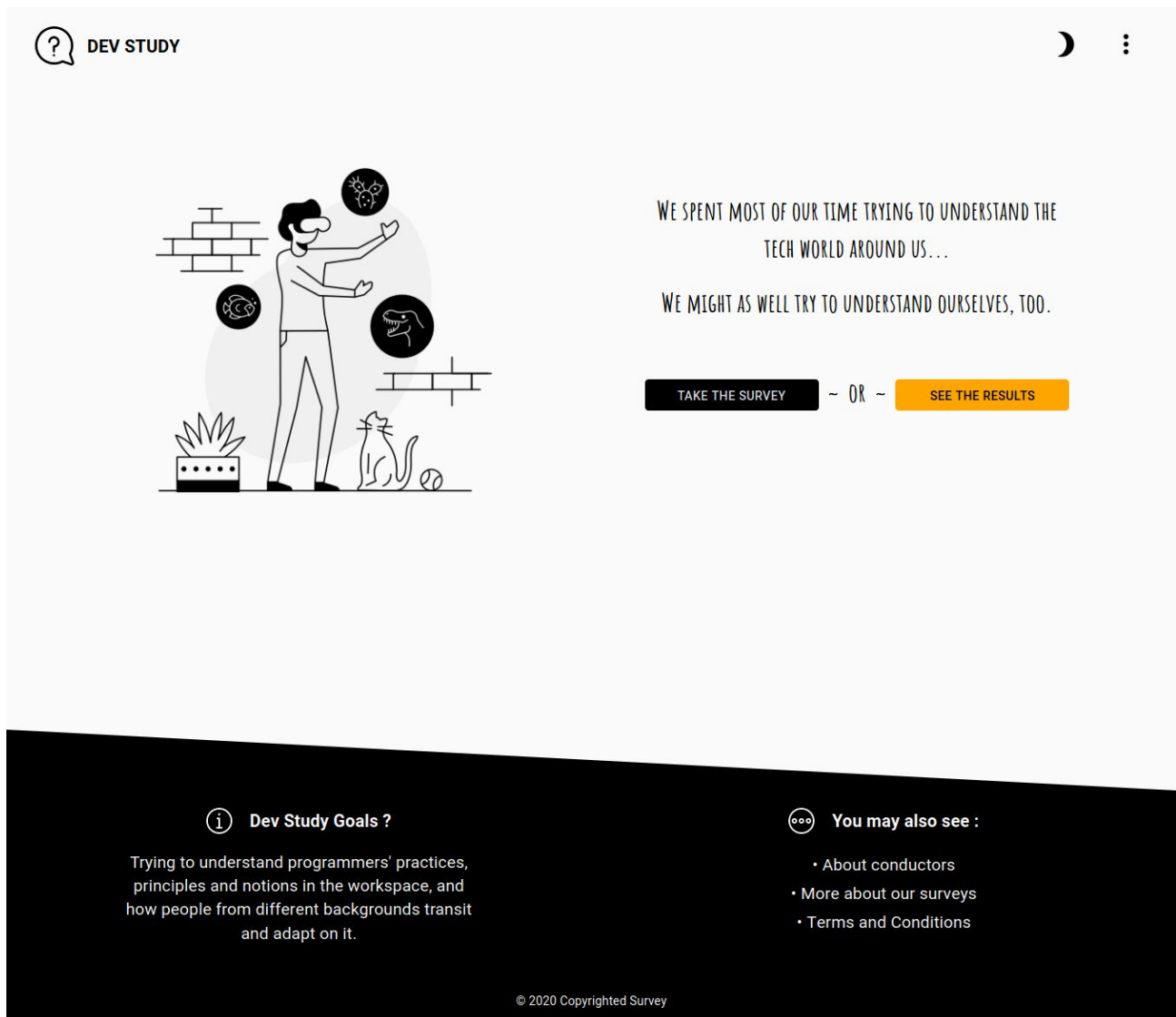


Figure 10: Dev-Study Use Case

3. SYSTEM DESIGN

In this chapter we will delve into the designs of the survey building and answer visualization. We will try to understand how every one of the core components of the survey is built and designed, what the patterns are and techniques selected, in order to clarify how the whole process is being functional and performant. Design phase was the first phase of this project life-cycle. It took approximately 1 month until it was well-established as how the different components will communicate and composite. The top priority was to design a tool that minimizes the repetition of the different kinds of processes, like questions that are being shared among many surveys and need to get edited, and also be as simple to understand and comprehend as possible. The Design Analysis chapter abstracts away the coding and technology details as much as possible, trying to structure our service in a language-agnostic way (Of course there are plenty of coding examples so that readers can fully grasp and embody the designs into a practical manner). The questions, surveys, survey answers follow the specifications and the format that we decided and the different procedures, too. We will guide you throughout the design phase of the implementation, starting with the cornerstone of the survey building, the questions.

3.1 Questions

3.1.1 Design

Questions are the building stone of the survey building. Surveys consist of questions that are dynamic throughout the survey life-cycle and that can be shared among different surveys. Different types of questions serve different kinds of meanings and each question targets a specific target group. For these reasons, the question creator should be able to select the appropriate question type for their intentions, to have a way to provide help to the user with different kinds of hints or restrict the user answers by applying a different set of rules. All these should be flexible and clearly described in a language-agnostic manner, so that any kind of developing environment can handle them. So, we came up with the following question specification, that we will see step by step.

```

{
  "id" : <Q_ID>,
  "type" : <String>,
  "text" : <String>,
  "values" : <Array of Strings>,
  "hint" : <String>,
  "rules" : <Array of Strings>,
  "answer" : <String>
}

```

Figure 11: Question Format

Each question is represented in JSON format. We picked that format, because it is widely accepted in the development community, especially when writing in a full Javascript environment. Let's analyze each one of question keys:

1. **id**: It is the unique question identifier that distincts it from the others.
2. **type**: Defines the question type, that suits the creators intentions. Possible types:
 - i) *SetOfStrings*: When the question can have multiple answers.
 - ii) *Number*: When the question answers is a numeric value.
 - iii) *String*: When the question answer is a single string.
3. **text**: This attribute contains the question text.
4. **values**: In this attribute, the user can add an array of values, for e.g. the *SetOfStrings* questions.
5. **hint**: The user can add a hint to a question, helping them to give more accurate answers.
6. **rules**: The question rules define the possible restrictions that can be applied to the question, like:
 - i) *Float*: Restricting number question answers to float values only.
 - ii) *Integer*: Restricting number question answers to integers values only.
 - iii) *Scale-n*: The <0-n> scale so a user can choose to number questions
 - iv) *Min-n*: The minimum number of answers that a person can choose
7. **answers**: The answer to the question

As it can be seen, the above attributes depict the essence of a question and can be scaled to more question types, rules or other kinds of attributes.

Questions are bought and stored in our front and back-end systems with the above JSON format described.

3.1.2 Modeling

Questions usually remain with the above json format for the different needs of our app, like the question edit etc. But there are specific use cases, that questions need to be modeled to a class-like structure, in order to contain a set of functionality and properties. For example, the front-end needs a question class object to be instantiated in order to render a question, or the survey building process stores the questions class objects within each structure and not the JSON format. It is important to describe how the questions are modeled on our systems in an Object Oriented Manner. Let us start with a Class UML diagram:

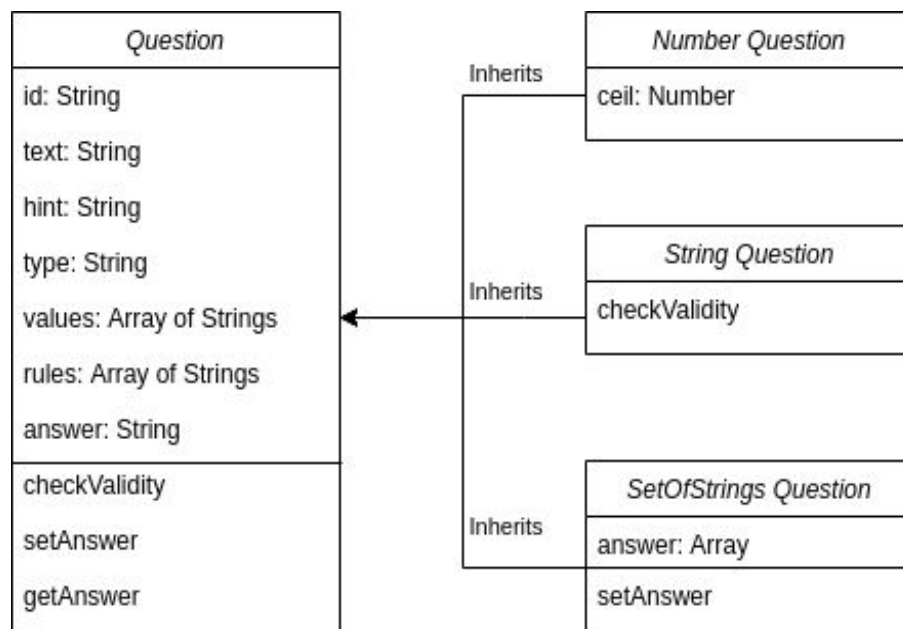
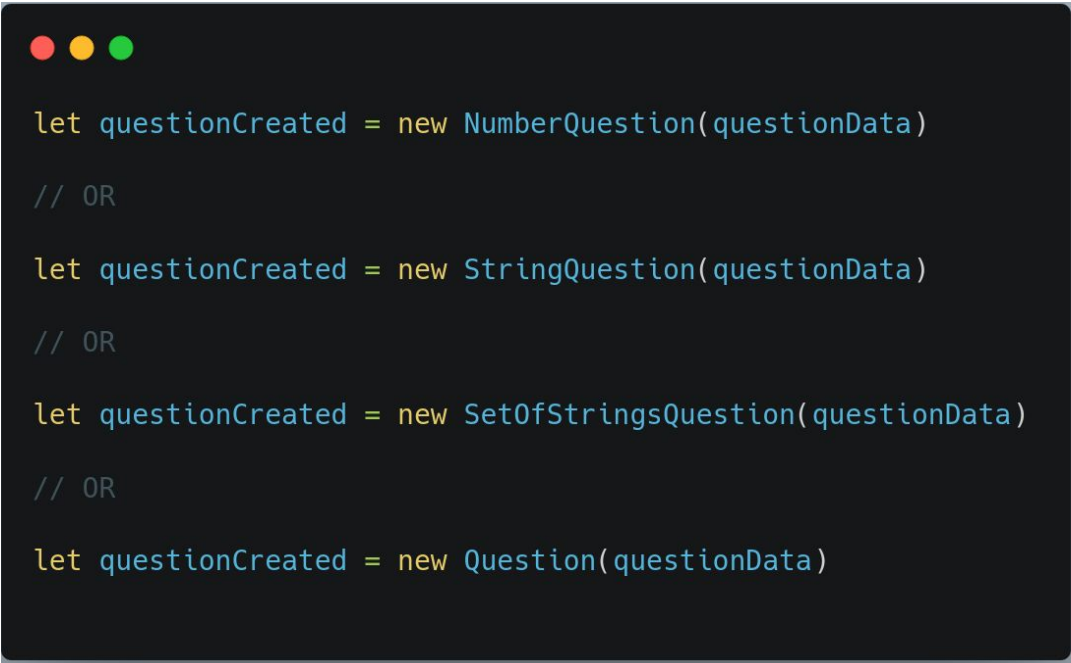


Diagram 1: Question Modeling

As seen on the above class diagram, we have the base question class that contains the information designed in the specification file. The base question fills each property on the constructor from question data given to it (a filled question spec file), along with the

setter/getters for the question answer and extra methods like `checkValidity` that validates whether the question's answer abides the set of rules or the type specified. From the question base class, the different types of questions derive their properties and methods by inheriting from it. Each question class of type X, that is the child of the question class, overloads functions like `check validity`, properties like `set answer` and `answer` (for example the set of strings question contains an array of answers given by the user), or adds more properties to the base class like a `ceil prop` for the number class. In the sequel of our previous statement, the scalability of the question designs extends to the data modeling of the question. The programmers can add more question types and their functionality by inheriting the question base class, taking advantage of the object oriented characteristics that the appropriate OOP languages provide. Our implementation follows the above diagram in Javascript. Let's start by showcasing how the developer invokes the creation of a new question based on the class model described, and we will then proceed in showing the code base for the question data modeling with oop possibilities of JS.



```
let questionCreated = new NumberQuestion(questionData)
// OR
let questionCreated = new StringQuestion(questionData)
// OR
let questionCreated = new SetOfStringsQuestion(questionData)
// OR
let questionCreated = new Question(questionData)
```

Code 1: Question Creation Code

```

class Question {
  constructor(questionData) {
    this.id = questionData.id
    this.text = questionData.text
    this.hint = questionData.hint
    this.type = questionData.type
    this.values = questionData.values
    this.rules = questionData.rules
    this.valid = false
    this.error = null
    this.answer = null
  }

  checkValidity(value) {
    if (!value) this.valid = false
    return null
  }

  setAnswer(value) {
    this.error = this.checkValidity(value)
    this.valid = this.error === null
    this.answer = value
  }
}

class NumberQuestion extends Question {
  constructor(questionData) {
    this.answer = null
    for (const rule of this.rules)
      if (rule.startsWith("Scale"))
        this.ceil = parseInt(rule.split("-")[1])
  }
}

class StringQuestion extends Question {
  constructor(questionData) {
    this.answer = null
  }

  checkValidity(value) {
    if (!value) return "Please answer this question"
    // Integer Question
    if (this.rules.includes("Integer")) {
      let str = value.trim()
      str = str.replace(/^0+/, "") || "0"
      const n = Math.floor(Number(str))
      if (n !== Infinity && String(n) === str) return null
      return "Answer should be an Integer"
    }
    // Float Question
    if (this.rules.includes("Float")) {
      if (!isNaN(value) && +value >= 0 && +value <= 10) return null
      return "Answer should be a Float from 1 to 10"
    }
    return null // NULL indicates that there wasn't an error
  }
}

class SetOfStringsQuestion extends Question {
  constructor(questionData) {
    super(questionData)
    this.answer = []
  }

  setAnswer(value) {
    this.error = this.checkValidity(value)
    this.valid = this.error === null
    const answerIndx = this.answer.indexOf(value)
    // If is already checked
    if (answerIndx !== -1) this.answer.splice(answerIndx, 1)
    else this.answer.push(value)
  }
}

```

Code 2: Question Class Code

3.1.3 Rendering

Each question can be rendered on our application, by first creating a question class object, as seen on the above examples. Each front-end framework that wants to render the question to the browser requires a question renderer, that based on the question type, renders the right type of question. For each question type, a different form should be displayed, for example SetOfStrings questions is a checkbox, String questions with multiple values should be represented by a set of radio buttons etc. Each question form for the different question types is a front-end component and can be based on any framework. Developers can create their own UIs for the questions that will be rendered. The only critical part is to connect the onInputChange functionality. Whenever a rendered question changes its input value, which happens when a user selects an input or types an answer, should call the innate way of question component to handle on input changed events and call the question's class object setAnswer, in order to update the answer within our question objects. Let's start by showcasing a pseudo code and design example of how our survey renderer works:

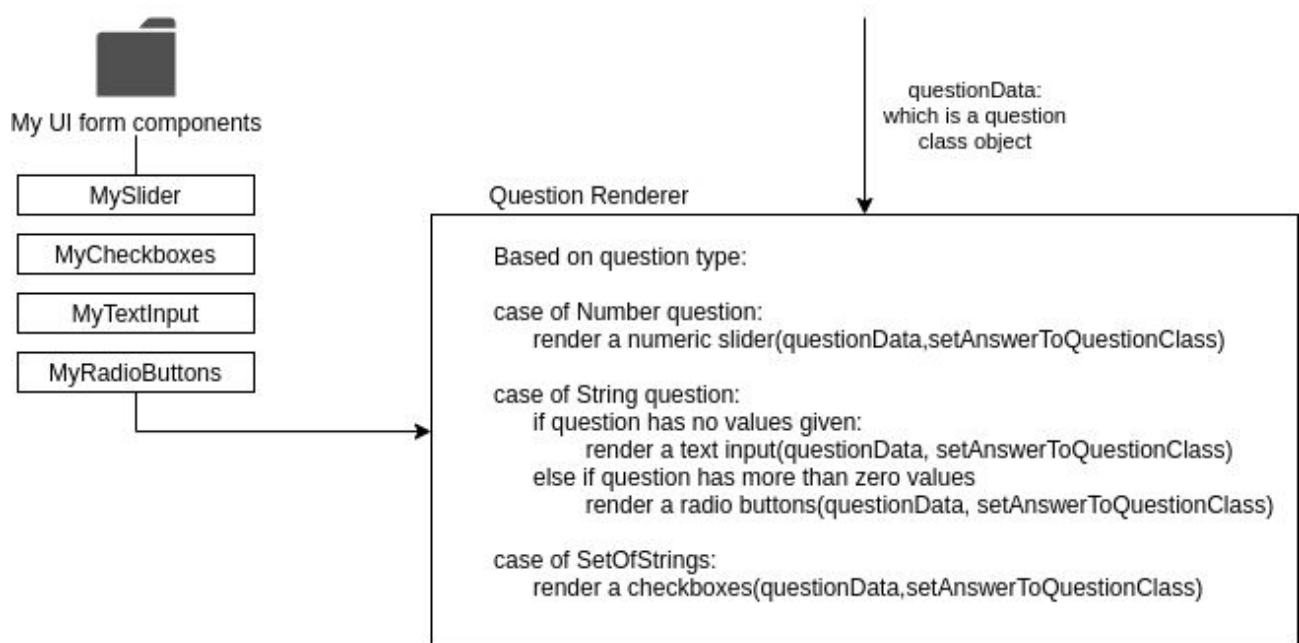


Diagram 2: Question Rendering

The question renderer contains the conditional logic described. For the different question types, render the right form components, that are implemented in a different directory. By that way, developers can add different directories with their own UI as already mentioned.

Before showing real code examples from our implementation, we will sum up the question rendering process. Let's see another design, outlining the process:

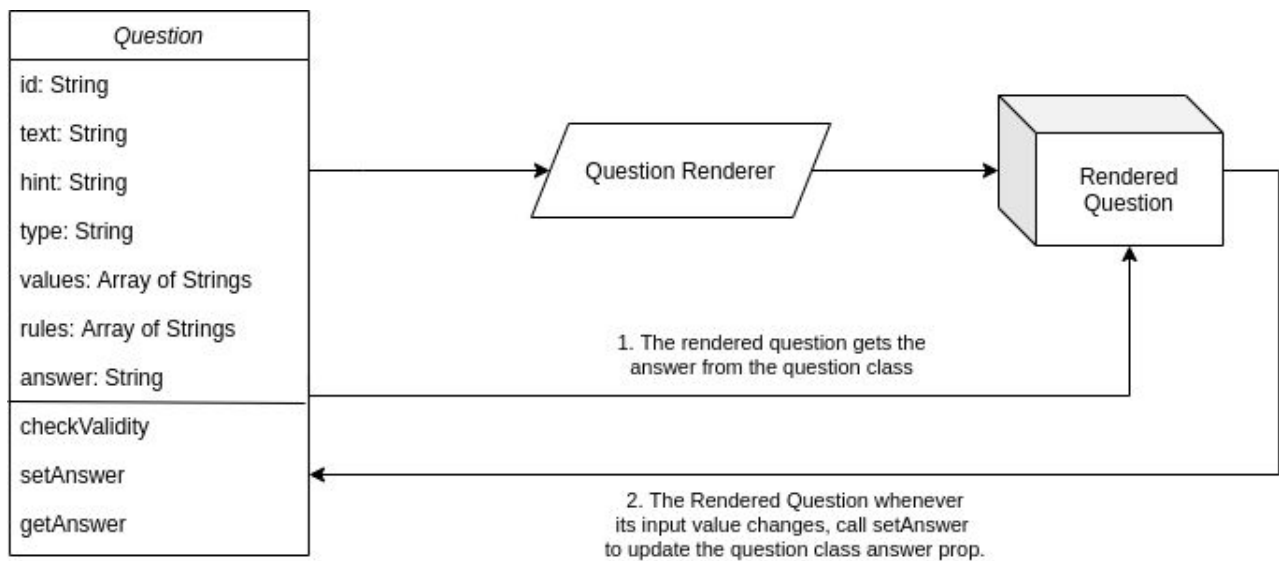


Diagram 3: Question Rendering Abstract

The Question is the object class that is passed to the question renderer. The question renderer produces the rendered question. It is important, for once again, to highlight that **the rendered question gets its answer from the question class, and connects its innate event handler with the question's setAnswer method, to update it** whenever the answer changes from the form input. A question component can be implemented by the following way, which is a real-code example from our implementation. Take a look on the **onChange** property, which is the event listener that will call the setAnswer. (The code is written in ReactJs and imports the MaterialUI for the checkbox. More about the technical details can be found on the 3rd chapter: Technical Analysis).

```

import React from "react"
import Checkbox from "@material-ui/core/Checkbox"

export default function MaterialCheckboxes(props) {
  const questionConfig = props.questionData
  return (
    <div className="myStyle">
      {questionConfig.values.map((option, index) => (
        <div>
          <FormLabel
            label={option}
            control={
              <Checkbox
                className={classes.checkbox}
                id={questionConfig.id + index}
                name={questionConfig.id + index}
                checked={
                  questionConfig.answerAlreadySelected()
                }
                value={option}
                onChange={props.changed}
              />
            }
          />
        </div>
      ))}
    </div>
  )
}

```

Code 3: Material Checkbox Form Component

The question form components look like the above implementation and the question renderer that imports and uses them looks like the following image.

```

import React from "react";

import {
  RadioButton,
  TextInput,
  Dropdown,
  Checkbox,
  Slider,
} from "../../FormComponents/MaterialUI/MaterialUI";

const question = (props) => {
  let inputElement = null;
  const questionConfig = props.questionData;
  /* Based on the question type create the proper input */
  switch (questionConfig.type) {
    case "Number":
      inputElement = (
        <Slider questionData={props.questionData} changed={props.changed} />
      );
      break;
    case "String":
      if (questionConfig.values.length === 0)
        inputElement = (
          <TextInput
            questionData={props.questionData}
            changed={props.changed}
          />
        );
      else if (questionConfig.values.length < 5)
        inputElement = (
          <RadioButton
            questionData={props.questionData}
            changed={props.changed}
          />
        );
      else
        inputElement = (
          <Dropdown
            questionData={props.questionData}
            changed={props.changed}
          />
        );
      break;
    case "SetOfStrings":
      inputElement = (
        <Checkbox
          questionData={props.questionData}
          changed={props.changed}
        />
      );
      break;
    default:
      inputElement = <p> No Valid Question Type </p>;
  }
  return (
    <div className="surveyQuestion">
      <p className="questionLabel">
        {questionConfig.text}
      </p>
      {inputElement}
      <div className="questionErrorText">
        <p> {questionConfig.error} </p>
      </div>
    </div>
  );
};

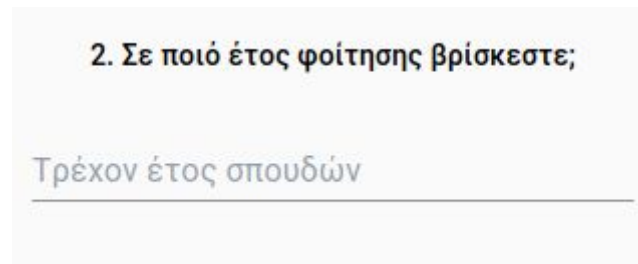
export default question;

```

Code 4: Question Renderer Code

Examples

For the last part of the Question Rendering, let us present some rendered questions and see how they look on our webpage:

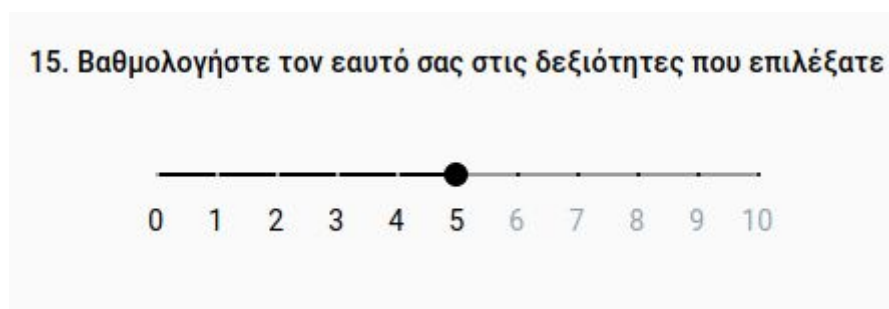


2. Σε ποιό έτος φοίτησης βρίσκεστε;

Τρέχον έτος σπουδών

A text input field with the placeholder text "Τρέχον έτος σπουδών" (Current year of study).

Figure 12: Text Input Question

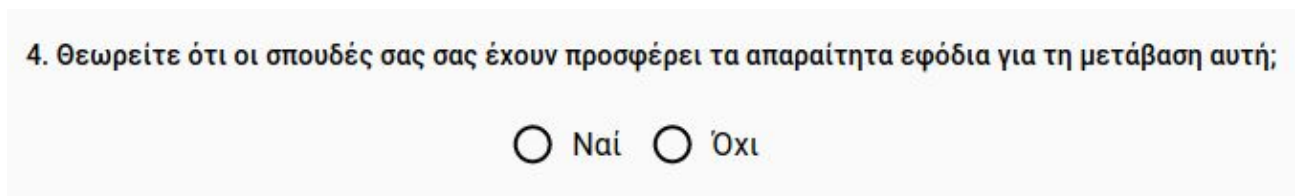


15. Βαθμολογήστε τον εαυτό σας στις δεξιότητες που επιλέξατε

0 1 2 3 4 5 6 7 8 9 10

A horizontal slider with a range from 0 to 10. The slider is currently set to 5.

Figure 13: Slider Question

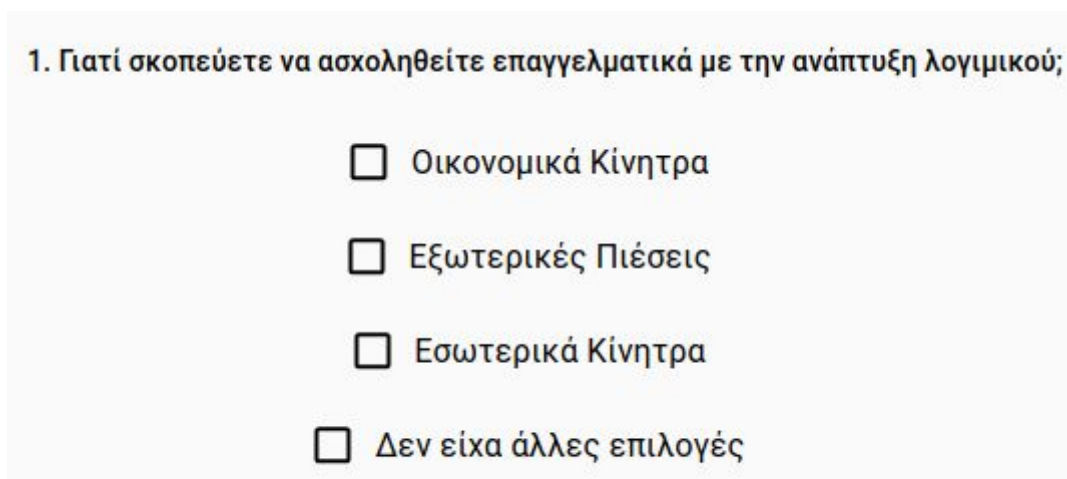


4. Θεωρείτε ότι οι σπουδές σας σας έχουν προσφέρει τα απαραίτητα εφόδια για τη μετάβαση αυτή;

☐ Ναι ☐ Όχι

Two radio buttons labeled "Ναι" (Yes) and "Όχι" (No).

Figure 14: Radio Button Question



1. Γιατί σκοπεύετε να ασχοληθείτε επαγγελματικά με την ανάπτυξη λογισμικού;

☐ Οικονομικά Κίνητρα

☐ Εξωτερικές Πιέσεις

☐ Εσωτερικά Κίνητρα

☐ Δεν είχα άλλες επιλογές

Four checkboxes with labels: "Οικονομικά Κίνητρα" (Economic Incentives), "Εξωτερικές Πιέσεις" (External Pressures), "Εσωτερικά Κίνητρα" (Internal Incentives), and "Δεν είχα άλλες επιλογές" (I had no other options).

Figure 15: Checkbox Question

3.2 Surveys

We will now move on to the core component of our service, the survey. We will follow the question analysis structure, starting with the Survey design and later move on Survey modeling and rendering for the platform usage.

3.2.1 Design

The survey designs exists in 2 formats:

- 1) The storage format.
- 2) The run-time format.

3.2.1.1 Storage Format

We will start off with the storage format. How the **survey is stored in our systems**.

```
{
  "id" : <String>,
  "title" : <String>,
  "shortDescription" : <String>,
  "longDescription" : <String>,
  "icon" <String>,
  "dateCreated" : <String>,
  "sections" : <Array of Sections>
}

Section: {
  "id" : <String>,
  "title" : <String>,
  "description" : <String>,
  "questions" : <Array of Strings>
}
```

Figure 16: Storage Format Structure

The survey is structured in a JSON format. It contains the basic fields, like id, title, short and long description, the survey icon and date created, that describe the survey. The section field, nests a json object which is the section format. The section contains the id, title and description of the section along with the questions field. The question field is an array of strings, which are basically the ids of the questions contained in the survey. The section and question order follows the indexing within the structure.

An example of a stored survey within our systems looks like:

```
{
  "id" : "STUD",
  "title" : "Ερωτηματολόγιο Φοιτητών",
  "icon" : "...",
  "longDescription" : "...",
  "shortDescription" : "...",
  "dateCreated" : "...",
  "sections" : [
    {
      "id": "TRNST",
      "title" : "Σχολή και Μετάβαση στην αγορά εργασίας",
      "description" : "Οι παρακάτω ερωτήσεις...",
      "questions" : ["Q01", "Q02", "Q03", ...]
    }, ...
  ]
}
```

Figure 17: Example of Storage Format

We decided the above format, as it is the minimal possible way to represent survey contents and structure, while retaining its scalability.

3.2.1.2 Runtime Format

The above format is useful for describing the survey and storing it in the data storing technologies but on run-time it is not so performant. Let us see how the run-time format of the survey looks like and we will justify why we made 2 versions for the survey depiction, along with the outline of their conversions.

```

{
  "id" : "STUD",
  "title" : "Ερωτηματολόγιο Φοιτητών",
  "icon" : "student.svg",
  "longDescription" : "Το ερωτηματολόγιο απευθύνεται...",
  "shortDescription" : "Για φοιτητές...",
  "dateCreated" : "26/09/2020",
  "sections" : [
    {
      "id" : "TRNST",
      "description" : "Οι παρακάτω ερωτήσεις ...",
      "title" : "Σχολή και Μετάβαση...",
      "questions" : [
        {
          "answer" : "",
          "values" : [
            "Οικονομικά Κίνητρα",
            "Εξωτερικές Πιέσεις",
            "Εσωτερικά Κίνητρα"
          ],
          "hint" : "Μπορείτε να επιλέξετε...",
          "rules" : [ ],
          "id" : "Q01",
          "text" : "Γιατί σκοπεύετε...;",
          "type" : "SetOfStrings"
        }, ...
      ]
    }
  ]
}

```

Figure 18: Runtime Format

This survey format is almost identical to the previous one. The difference is that instead of storing an array of question ids in each section, we store the whole question data. That creates a bigger survey format but really helps the survey creation process. Let us see how.

3.2.1.3 Difference between the two survey formats

The question is, why don't we use one and only survey format? Here are the reasons:

- The storage format, where the section questions are represented by a set of survey ids, is a compact and minimal way to describe the survey. Also, when a question is updated, the survey will always contain the updated version, as it only stores the id of the question and not all of its data.

- The run-time format, where the section questions are represented by their whole specifications, is better for performance reasons. As we already stated, in order to render a question, the question renderer needs a question object that requires all of the questions data. If we stored the survey the storage way, then for each question we would need to search the question pool to find its data, because we would only have the id. In contrast, the run-time format provides all the question information when the survey is about to be created, without having to search for the questions data.

3.2.1.4 Conversion between the two formats and outline

The last thing that needs to be clarified is how the whole process integrates and functions together. Descriptively, we start off by building the survey in the storage-format. When adding a question, the question id is being pushed in the section questions array, and step by step, the survey is completed. The question and section order is being handled by the survey builder that we showcased in the first chapter. Upon survey creation, the survey specification file is stored on the back-end. Whenever a question is being edited or the survey updated, the spec. file changes on the backend and stays there with the same format. The conversion of the survey specification file to the run-time format happens when the survey needs to be rendered. The backend gets a request to render the survey with a specific id, get the specification file of the storage format and convert to the run-time. From then, it passes it to survey creation and rendering for it to happen. Let's project an outline of the survey format conversion to understand it.

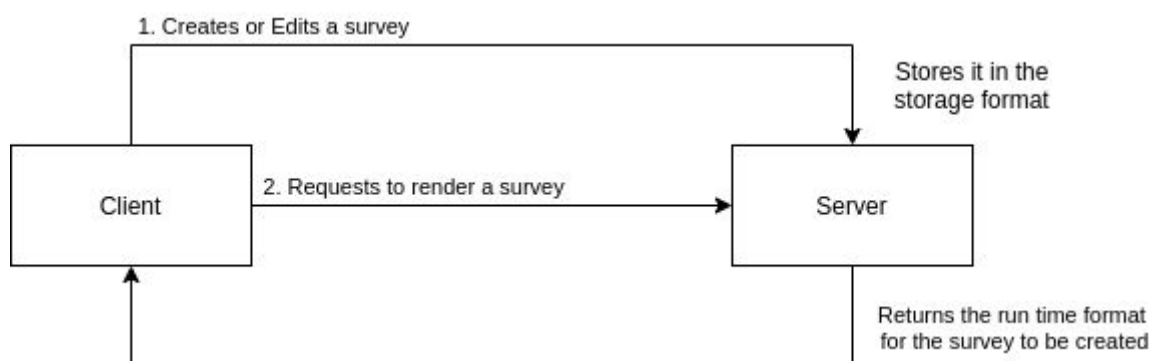


Diagram 4: Conversion Between Survey Formats

3.2.2 Modeling

Like the questions, we need to model the survey, in order to use it on our services. A survey with the above specification needs to be translated to a class object, so that it can be rendered and take usage of the OOP possibilities. For once again, let us start with a class diagram for the survey and we will explain it thoroughly.

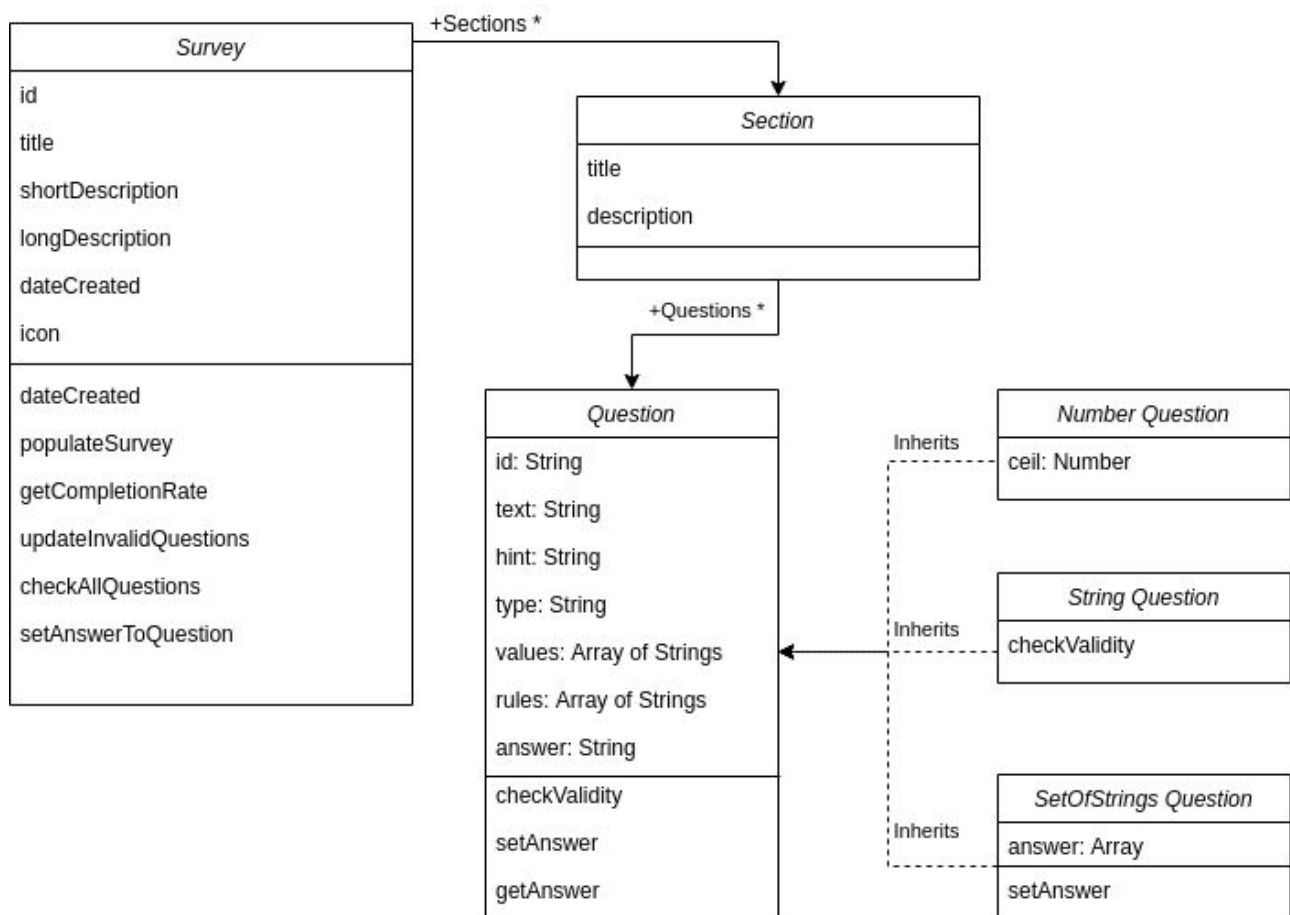


Diagram 5: Survey Class Model

The survey object contains all the required functionality. It's scalable, easy to understand and maintain and provides all the important information for the survey life-cycle. The class includes methods like `populateSurvey` from an already answered survey, `getCompletionRate` for the completion percentage of the survey, check the survey for questions with errors and update the invalid question list, `setAnswer` to a question of the survey etc. Each survey contains an array of section objects, and that array of sections

contain an array of question objects. **The survey class is instantiated given a survey in run-time format**, and keeps track of the survey state while this is being “alive”.

A survey object can be created, in code-level, like this:

```
// Create the survey class where we render questions and save answers
let surveyCreated = new Survey(survey_specification_file);
```

Code 5: Survey Creation

And the survey class, looks like this:

```
import { questionFactory } from "../QuestionClass"

class Section {
  constructor(title, description) {
    this.title = title
    this.description = description
    this.questions = []
  }
}

class Survey {
  constructor(questionnaire) {
    this.id = questionnaire.id
    this.title = questionnaire.title
    this.description = questionnaire.description
    this.sections = []
    this.invalidQuestions = []
    this.answeredQuestions = 0
    this.totalQuestions = 0
    // For each section in questionnaire, create an instance and add it to survey
    questionnaire.sections.forEach((section, sectionIndx) => {
      const newSection = new Section(section.title, section.description)
      // For each question in section, create an instance and add it to section
      section.questions.forEach((questionData, questionIndx) => {
        // Create key that helps to index a question within the survey
        const key = [sectionIndx, questionIndx]
        const newQuestion = questionFactory(key, questionData)
        newSection.questions.push(newQuestion)
        this.totalQuestions++
      })
      this.sections.push(newSection)
    })
  }

  // Export the answered survey with its meta-data
  exportSurvey(submissionTime, submissionDate) {
    /* export survey code */
  }

  // Populate survey with an already filled survey object
  populateSurvey(filledSurvey) {
    /* populate survey code */
  }

  /* The other methods declaration... */
}
```

Code 6: Survey Class

3.2.3 Rendering

The final phase of the survey design analysis is to lay out the rendering process. The survey form is actually the survey in a web form, that contains all the questions and gets the user answers. It gets as a parameter the survey specification file, in a run-time format, creates a survey object with it and starts to construct the survey. Let's analyze this step by step:

```
constructor(props) {
  super();
  // Create the survey class where we render questions and save answers
  this.surveyCreated = new DataModel.Survey(props.questionnaire);
}
```

Code 7: Survey Rendering - Survey Creation

First, create the survey object based on the specification file, as explained. Let us explain the rendering process now.

```
renderQuestion = (question) => {
  return (
    <SurveyFormQuestion
      questionData={question}
      changed={(event) => this.inputChangedHandler(event, question)}
    />
  );
};

renderSection = (section) => {
  return (
    <div className="surveySection">
      <header>
        {section.title}
      </header>
      <p>
        {section.description}
      </p>
      {section.questions.map((question) =>
        this.renderQuestion(question)
      )}
    </div>
  );
};

const Survey = (
  <div>
    {this.renderSection(
      this.surveyCreated[firstSection]
    )}
  </div>
)
```

Code 8: Survey Rendering - Methods

The survey rendering starts off by selecting a section to render. We will begin with the first section. The **renderSection** function gets a section object as a parameter and renders the Section info along with the questions that belong to it. For each question the renderSection function calls the **renderQuestion** that we described in the *2.1.3 Question Rendering* chapter. Here, it is important to mention again that to the renderQuestion it is passed the **inputChangedHanler** that connects the question events handler to **setAnswer**, in order to keep the survey updated.

```
inputChangedHandler = (event, question) => {  
  // set new answer in survey class  
  this.surveyCreated.setAnswerToQuestion(question, event.target.value);  
};
```

Code 9: Input Changed Handler

There are also extra components that improve the user experience, like the survey helper and section navigation. They are implemented in react, too, and care for the user's navigation between sections and questions. These and the whole question and section indexing is the sequence of the carefully designed survey structure, following the section and question order, making it easy to be searched.

All of the previous analysis will result in the following survey form which users can fill and submit. It guides them through the erroneous questions and stores a set of metadata about the survey process (like half-completed surveys, the time it took for the user to answer the survey, etc.).

Let us see an example of a rendered survey to visualize the results.

Ερωτηματολόγιο Φοιτητών

Το ερωτηματολόγιο απευθύνεται σε τελειόφοιτους τμημάτων πληροφορικής, οι οποίοι ενδιαφέρονται να ασχοληθούν επαγγελματικά με την ανάπτυξη λογισμικού (Software Engineering and Development).

Το well-being του προγραμματιστή [2/3]

Οι ερωτήσεις στο παρακάτω κομμάτι έχουν να κάνουν με τις συνήθειες και την υγεία του προγραμματιστή

1. Βαθμολογήστε την ποιότητα των συνηθειών σας (διατροφή, άσκηση, ποιότητα ύπνου...).

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☒ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

2. Είστε ευχαριστημένοι απο την κοινωνική σας ζωή;

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☒ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

3. Η επαγγελματική ενασχόληση κρίνετε πως θα τον επιβαρύνει;

☐ Ναι ☐ Όχι

4. Οι σπουδές σας πόσο τον άλλαξαν;

☐ Καθόλου ☐ Λίγο ☐ Αρκετά ☐ Πολύ

< Προηγούμενο
 Επόμενο >



Figure 19: Rendered Survey

Now, that we have finished the question and survey analysis and how the whole process results in a rendered survey, we will summarize the whole process with a big and detailed diagram.

Survey Rendering Summary

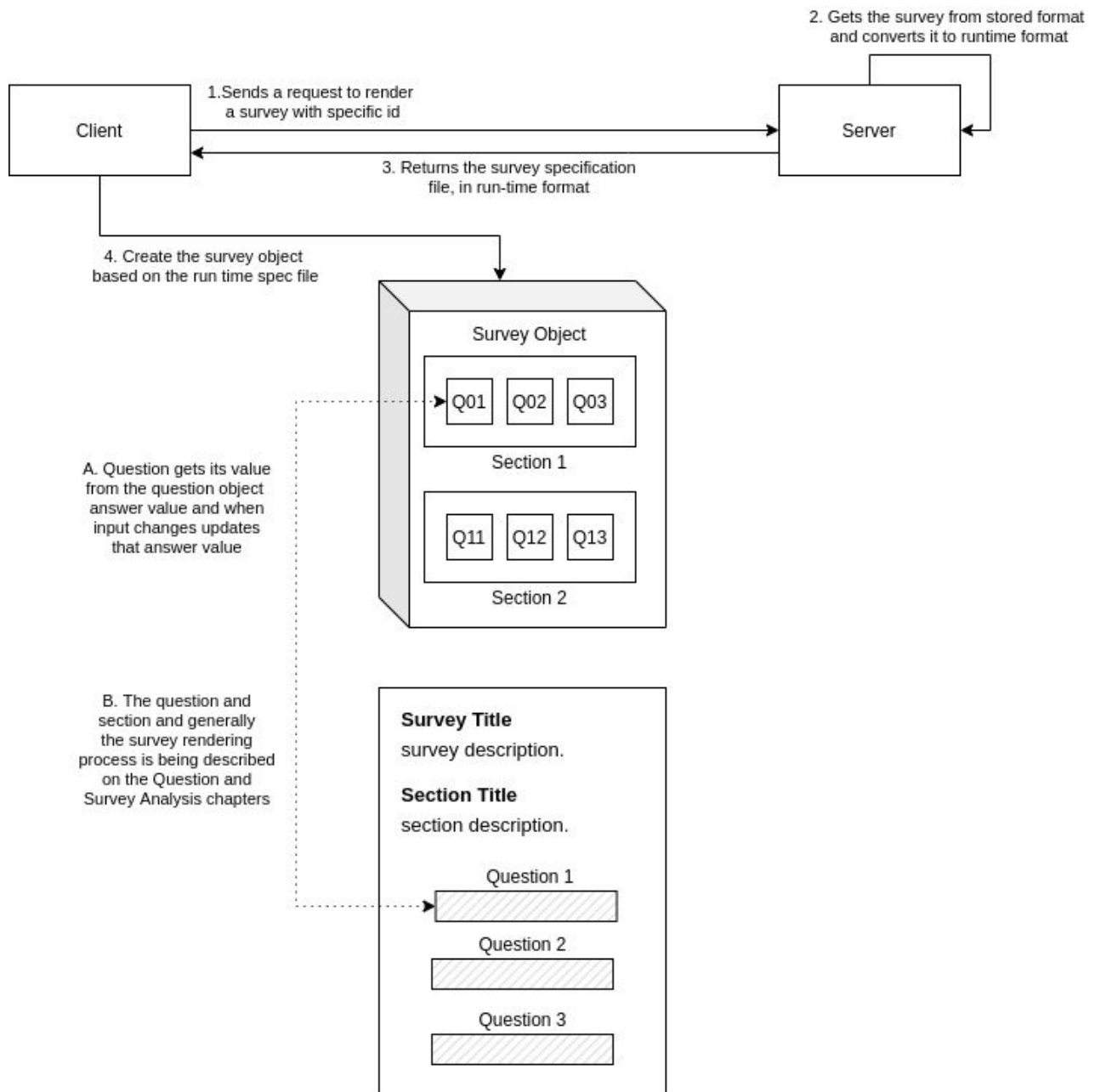


Diagram 6: Survey Rendering Abstract

With survey analysis finished, we can now move to the next analysis phase. The Answered Surveys analysis. Each survey, when completed, should store the survey answers in a digestible format for the visualization and data analysis to happen. In the following chapter we will describe the answered survey format and design, and why we decided on this format, while also explaining the whole process of an answered survey export from the survey object, storing to our back-end systems and retrieval to our client side.

3.3 Answered Surveys

We now proceed to the Answered Survey analysis, which are actually the answers of the surveys.

3.3.1 Design

The three building blocks of any survey are questions, the survey and the answers. In this chapter we will explain the last document structure that the survey building and report page is using, the answered surveys.

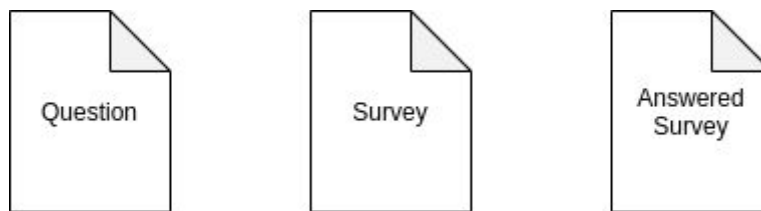


Figure 20: 3 Document Types

It is important to define a good format for the answered surveys. An answered survey format should let the system:

- Easily read the answers to any question
- Visualize the answers to any plot or graph
- Allow question combinations and aggregations
- Contain metadata about each survey

For all these reasons, we tried to approach how we handle an answered survey in real-world, with documents. Each answered survey is actually a document that contains the answers for each question, along with some metadata about it. Of course each answered survey contains the survey ID that it corresponds to and information like the time it took an individual to complete it, the date it was submitted and other kinds of fields that we will showcase. The last thing to mention is that each **survey's answers are represented by one document**, so that it can be unique and distinctable from other answered surveys and let analysts combine answers given by the same person. Of course in our systems we do not hold information and data about the participants. Each answered survey has only the answers to the question and other meta information, but there is no way to identify a person from an answered survey, making it fully anonymous.

Now, let us see how an answered survey looks like.


```

{
  "id" : "STUD",
  "took" : 4,
  "submittedOn" : "9/10/2020",
  "answers" : {
    "Q01" : [
      "Οικονομικά Κίνητρα",
      "Εξωτερικές Πιέσεις"
    ],
    "Q02" : "Ναί",
    "Q03" : [
      "Οι γνώσεις μου δεν είναι αρκετές για την αγορά εργασίας",
      "Δεν ξέρω ακριβώς τον τομέα πληροφορικής που θέλω να ασχοληθώ"
    ],
    "Q04" : "Όχι",
    "Q05" : [
      "Καλύτερη δικτύωση",
      "Περισσότερη φυσική παρουσία"
    ],
    "...",
    "Q28" : "Ναί",
    "Q29" : 8,
    "Q30" : 7,
    "Q31" : "Όχι",
    "Q32" : "Λίγο",
    "Q33" : "Τμήμα Πληροφορικής & Τηλεπικοινωνιών ΕΚΠΑ",
    "Q34" : "6",
    "Q35" : "7.6",
    "Q36" : "Άντρας"
  }
}

```

Figure 21: Answered Survey Example

So, given that a participant completes a survey, then it is stored in our systems with the above format. Once again, it is a JSON file that contains the basic fields about the completion information, like the time it took the user to complete the survey, the ID of the survey that the participant answered and the date that it was submitted. The answers field is an object that contains for each question the answer to it. This question is represented by its id and the answer is the value or values that the user gave, when submitting the form.

3.3.2 Exportation and Storage

Now that we have described the answered survey structure and format, let us delineate how the system produces the above document/spec. file and how it is stored within our systems.

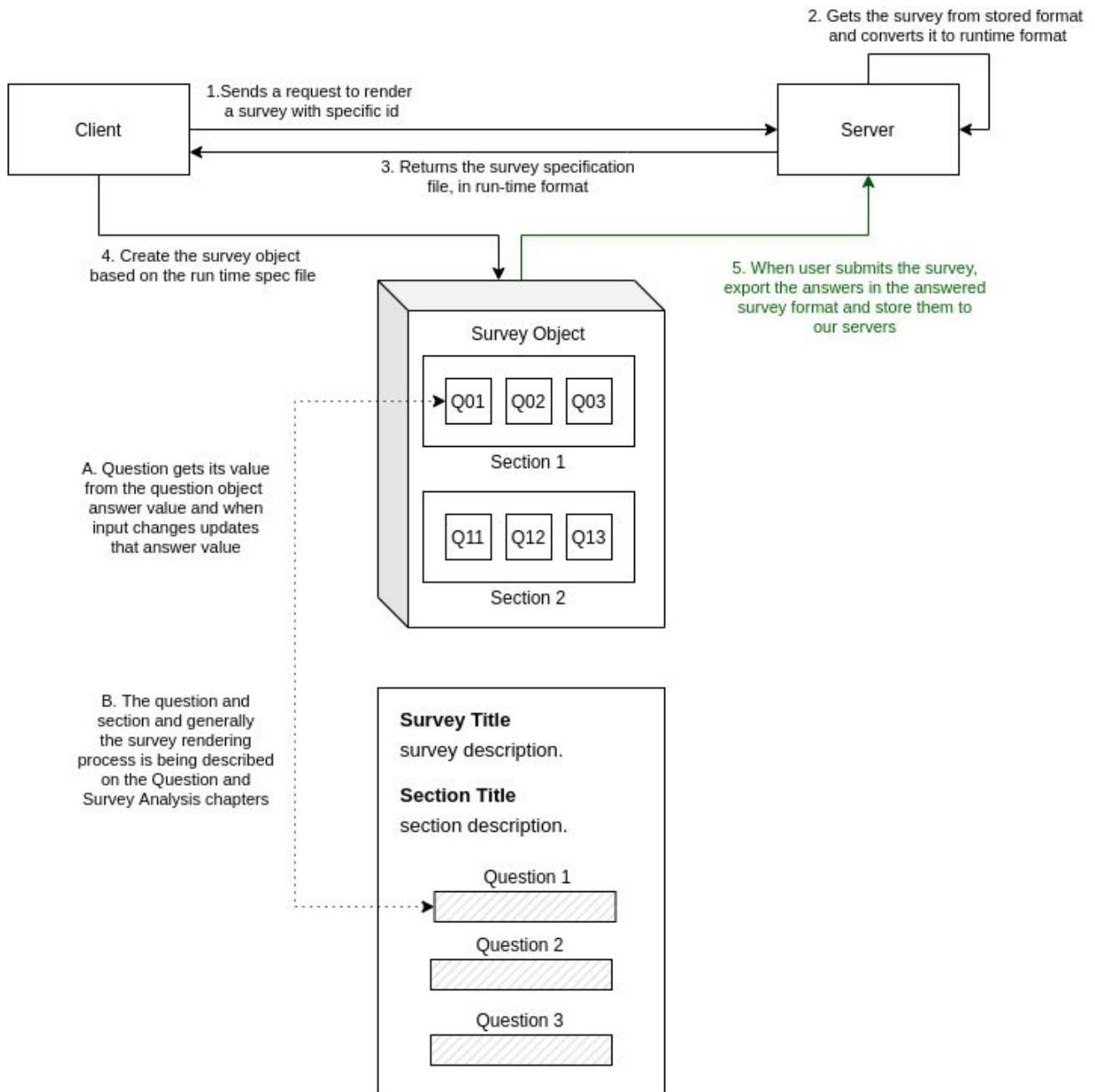


Diagram 7: Survey Exportation and Storage

In order to store the answered survey, we need to export the answers from the Survey Object that we explained in the Survey Rendering chapter.

The answers export to the answered survey format with the following sequence. When a user has requested to take a survey, our service renders the survey with the rendering process that we have already clarified. When they have completed the whole survey, they submit the survey and if that contains no errors, then the survey class object takes over in order to export the survey and store it in our systems. This happens in the following way:

```

submitSurveyHandler = (event) => {
  event.preventDefault();
  // check if form is valid
  const formIsValid = this.surveyCreated.checkAllQuestions();
  this.setState({ formIsValid });
  if (formIsValid) {
    const stopTime = performance.now();
    // Time it took user to complete the survey
    const elapsedTime = stopTime - this.startTime;
    // Store the date user took the survey
    const dateObj = new Date();
    const month = dateObj.getUTCMonth() + 1; // months from 1-12
    const day = dateObj.getUTCDate();
    const year = dateObj.getUTCFullYear();
    const currentDate = `${day}/${month}/${year}`;
    // Post the completed survey to backend
    axios
      .post(
        "/answered_surveys/",
        this.surveyCreated.exportSurvey(elapsedTime, currentDate)
      )
      .then((res) => {
        this.setState({ surveySubmitted: true });
      });
  }
};

```

Code 10: Survey Submission

As it can be seen in the above React implementation, when the user clicks the submit button, the submitSurveyHanlder comes and checks all questions for their validity and if the whole survey is valid then gets the last metadata and exports the survey answers with the survey object method. The exportSurvey method looks like this:

```
// Export the answered survey with its meta-data
exportSurvey(submissionTime, submissionDate) {
  const completedSurvey = {}
  completedSurvey.id = this.id
  // Submission time is on ms, save it on seconds
  completedSurvey.took = Math.ceil(submissionTime / 1000)
  completedSurvey.submittedOn = submissionDate
  const answers = {}
  this.sections.forEach((section) => {
    section.questions.forEach((question) => {
      answers[question.id] = question.answer
    })
  })
  completedSurvey.answers = answers
  return completedSurvey
}
```

Code 11: Export Survey Method

In a JS manner, the method for each section that exists in the survey, gets all of its question answers and creates the key-value pair, with the key the question id and value the answer. So the whole answered survey object is created by the above method, and is ready to be stored in our systems. The answered survey is sent to the backend with a POST method, but the technical details about the storage methods and communication between client and server will be explained in the Technical Analysis chapter.

We can now move to the next phase, that of Survey Reporting and Answers Visualization and Aggregation.

3.4 Results Visualization

3.4.1 General

The end-goal of each survey is to extract results about a central idea that you aim for or explore the data to see what else can be found. For these reasons, it is crucial to provide a way for the conductors to visualize the results and the answers given to them, while also offering an option to aggregate the questions, in order to reach new kinds of conclusions.

Now that we have explained the whole process of survey creation and storage to our systems, imagine that we have a whole load of answered surveys concerning a survey. The direct information that we can provide is the number of surveys completed, the average time it took users to submit them and the period the survey was more active. But an answered survey contains the critical part of our initial intentions, the answers.

Questions do have different types, meaning that the answers given to them are also of different types. Number questions are of number types, MutualExclusive types of questions contain distinct types of answers, while SetOfStrings also contains a different set of answers. This means that each question type requires a different kind of visualization and handling when it comes to presenting the results or processing them.

Furthermore, having different kinds of questions and a number of answered surveys, there should be a way to aggregate two or more questions, combine the results given and conclude new findings.

A survey can have a lot of questions. Easily navigating through them should be a priority to the user experience. Enhancing the user experience the graphs should be easy to understand. Also, our services give the option for anyone to download the whole survey reporting or specific question results, depending on their needs.

Last but not least, there are times that results and data are for anyone. Our platform gives the opportunity for the conductors to share the results with the public, allowing them to preview the results or have access to the Visual Wizard, so that they too can 'play around' with them.

All of the above statements are covered by the Results Visualization Analysis, where we unveil the underlying mechanisms of the visualization process along with the design artifacts that will help demystify how things work.

3.4.2 Results Reporting

For every question that we want to display, we need a simple piece of information about it. Based on the different kinds of questions, this simple piece of information changes. So far our Results Reporting services handle 2 kinds of questions when it comes to rendering (not to be confused with the question types) :

- i) Questions that we select a value based on a set of values, the *terms questions*.
- ii) Questions that we choose a specific numeric type, the *stats questions*.

Each kind of question needs a different type of handling when it comes to its visualization, in order for the results to be meaningful. Before analyzing these pieces of information and how the front-end renders them, I will lay out how the whole answer visualization happens.

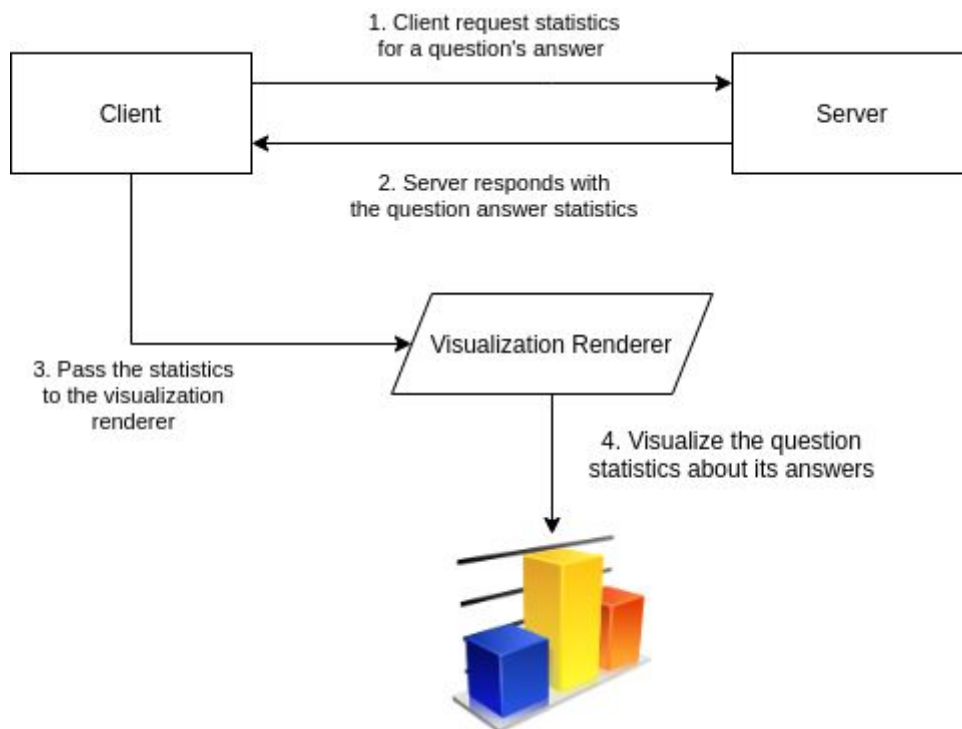


Diagram 8: Results Reporting - Server Client Interaction

The whole process of visualizing the question's answers can be summed up in 4 steps. The client requests the statistics for a question from the server, the server responds with the statistics and the client handles their visualization. The part of the process that handles the **result rendering** concerns the **front-end**, while the **statistics provided** happens to our **back-end**. Let us analyze this separation of concerns of these two parts and lastly integrate the whole solution for the question's answers visualization.

3.4.2.1 Results Rendering

Let us start from the front-end, which is responsible for the answer visualization. As mentioned, different question types are represented by different types of graphs. The piece of code that is responsible for rendering the right type of graph for the different question types is the **Visualization Renderer**. It applies the same conditional logic that the Question Renderer uses, keeping the implementation clean and simple. Let us see how the Visualization Renderer looks like theoretically:

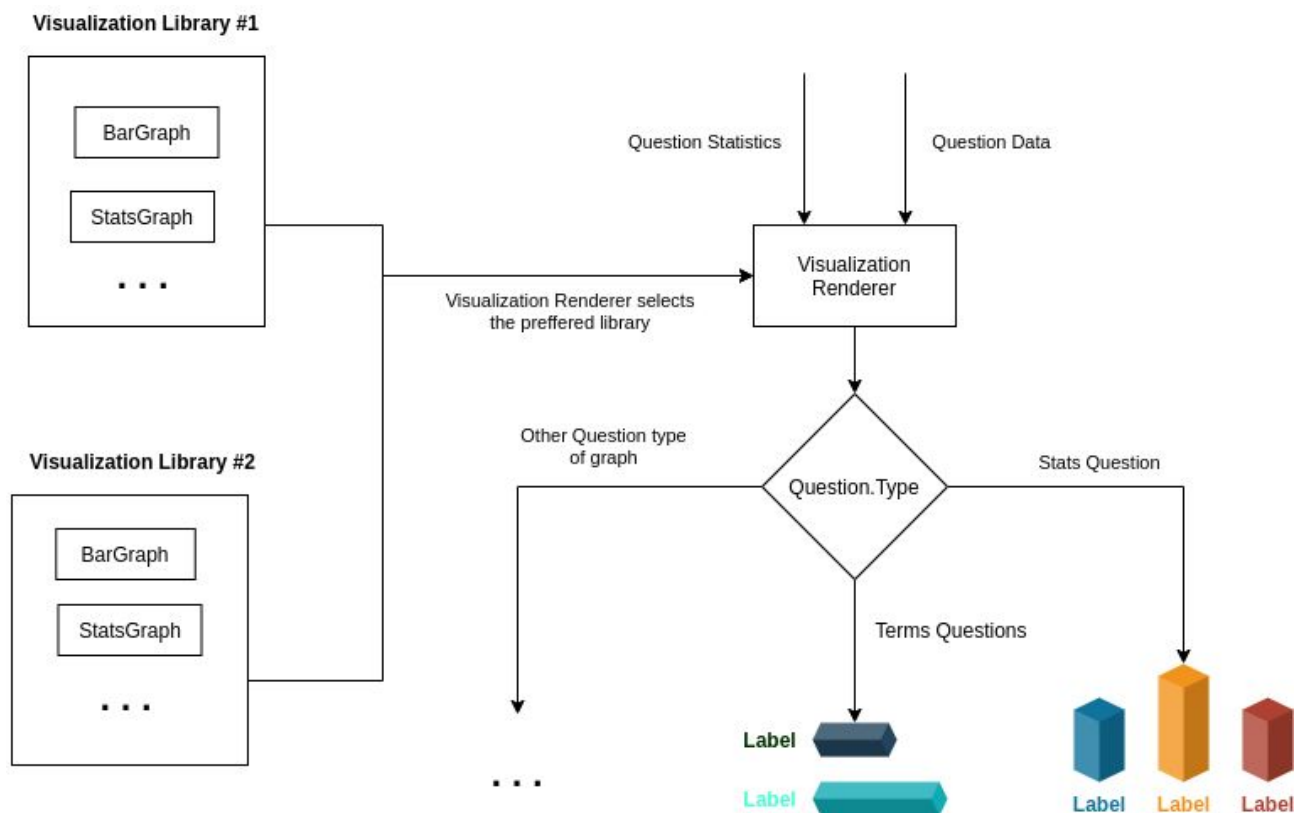


Diagram 9: Visualization Renderer

The visualization renderer needs 2 types of input. The statistics and data of the questions. Based on the question type, it renders the proper graph. The graphs that render those questions statistics, can be selected from the different visualization libraries that our code handles. The code logic and design will be explained in the Design Challenges chapter.

The Visualization Renderer implementation is the following:

```

import React from "react";
import { BarGraph, StatsGraph } from "../VisualizationComponents/RechartsUI/RechartsUI";
import "../VisualizationRenderer.css";

const visualization = (props) => {
  const questionData = props.questionData;
  const questionStats = props.questionStats;
  if (!questionData) return null;
  if (!questionStats) return null;
  var inputElement;
  /* Based on the question type create the proper visualization */
  switch (questionData.type) {
    case "Number":
      inputElement = <StatsGraph questionStats={questionStats.stats} />;
      break;

    case "SetOfStrings":
      inputElement = <BarGraph questionStats={questionStats.stats} />;
      break;
    default:
      inputElement = <BarGraph questionStats={questionStats.stats} />;
  }
  return <div>{inputElement}</div>;
};

export default visualization;

```

Code 12: Visualization Renderer

As we have already mentioned, there are two kinds of questions when it comes to rendering. The *terms questions* and *stats questions*. The question type Number is considered a stat question and all the other types of questions are considered term questions. For those two types of questions we have two different statistic formats and graphs. Let us see them:

A. Terms Questions

The terms questions statistics look like the following:

```

questionStatistics: {
  total_answers: 7,
  stats: [
    key: "Answer #1", doc_count: 4,
    key: "Answer #2", doc_count: 3
  ]
}

```

Figure 22: Term Questions Statistics Format

The terms questions consist of a set of possible answers to questions, in a string format. For the different answers the participants have given, we hold the label as the label/key value for the graph and doc_count represents the number of participants selected in that answer. These stats for the question's answers, along

with the total number of answered surveys, are sufficient for our implementation to draw this kind of graph:

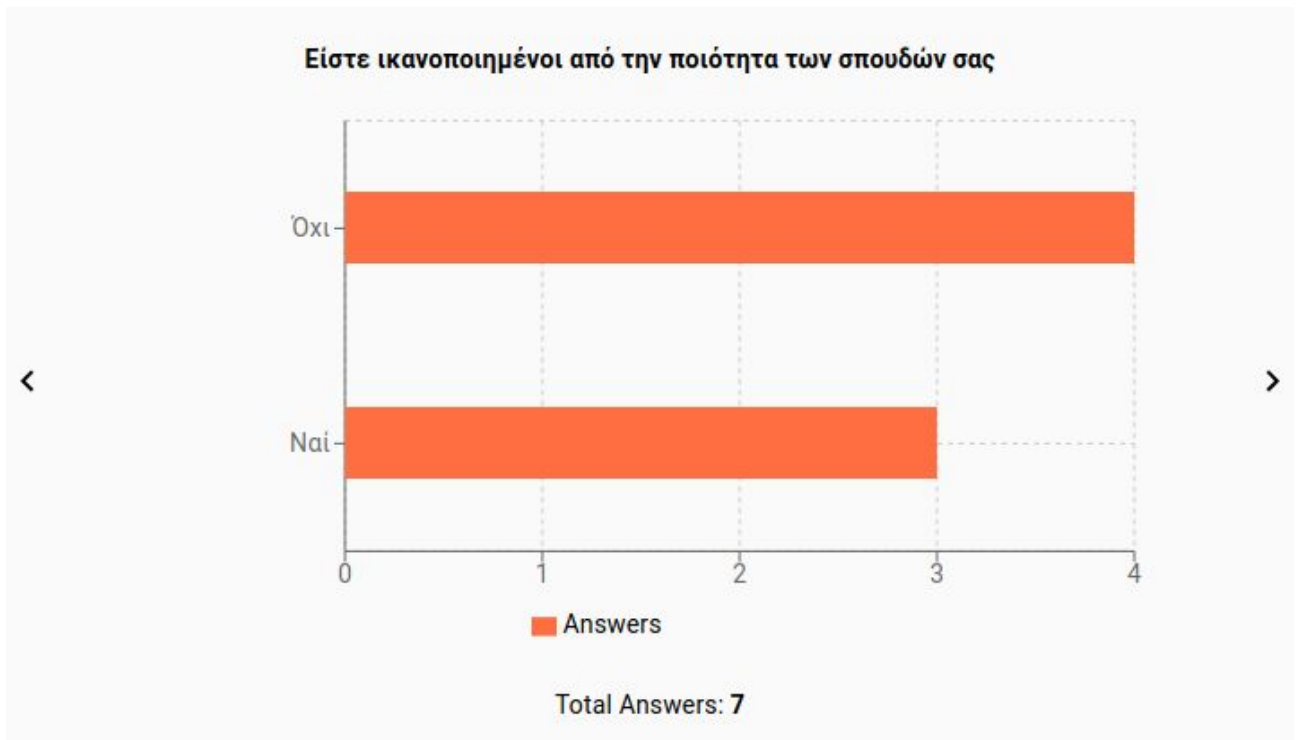


Figure 23: Rendered Terms Question Graph

The key in stats array represents the label of the graph, and doc-count the bar graph value. The above and the the following graphs are provided by the RechartsUI library, which we will go through in the technical analysis chapter. The above code implementation, written in React and using the library, is:

```

import React from "react";
import "recharts";
import "./BarGraph.css";

export default function BarGraph(props) {
  const questionStats = props.questionStats;
  const data = [];

  questionStats.map((questionStat) => {
    let graph_stat = {};
    graph_stat["name"] = questionStat.key;
    graph_stat["Answers"] = questionStat.doc_count;
    data.push(graph_stat);
  });

  return (
    <BarChart width={600} height={300} data={data}
      barGap={3} layout="vertical" barCategoryGap="30%"
    >
      <CartesianGrid strokeDasharray="3 3" />
      <XAxis type="number" />
      <YAxis width={150} dataKey="name" type="category" />
      <Tooltip />
      <Legend />
      <Bar maxBarSize={40} dataKey="Answers" fill="#ff6e40" />
    </BarChart>
  );
}

```

Code 13: Terms Questions Graph

B. Stats Questions

The Stats question format concerns actually only the Number questions. Because the answer to those questions is a numeric type, we would only need the core information which would allow us to depict the question essence:

```

questionStatistics: {
  total_answers: 7,
  stats: [
    avg: 4,
    count: 7,
    max: 7,
    min: 4,
    sum: 28
  ]
}

```

Figure 24: Stats Question Data Format

The statistics provider that will soon be explained, returns the above basic stats. We select to visualize only the avg that stands for the numeric average of the answers and the min/max values. The Stats Graph looks like this:

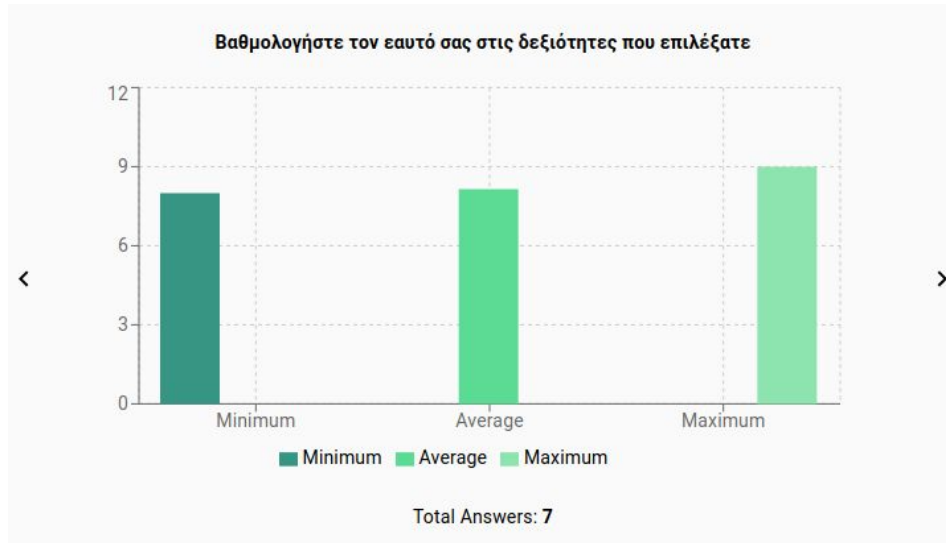


Figure 25: Rendered Stats Question Graph

```
import React from "react";
import "recharts";

export default function BarGraph(props) {
  const questionStats = props.questionStats;

  const data = [
    {
      name: "Minimum",
      Minimum: questionStats.min,
    },
    {
      name: "Average",
      Average: questionStats.avg,
    },
    {
      name: "Maximum",
      Maximum: questionStats.max,
    },
  ];

  return (
    <BarChart width={600} height={300} data={data}>
      <CartesianGrid strokeDasharray="3 3" />
      <XAxis dataKey="name" />
      <YAxis />
      <Tooltip />
      <Legend />
      <Bar barSize={45} dataKey="Minimum" fill="#379683" />
      <Bar barSize={45} dataKey="Average" fill="#5cdb95" />
      <Bar barSize={45} dataKey="Maximum" fill="#8ee4af" />
    </BarChart>
  );
}
```

Code 14: Stats Question Graph

Given the Stats type of questions we have finished the Results Rendering documentation. Of course given the code scalability, it is easy to add different kinds of graphs for different question types. For whatever statistics and information we have, the front-end can handle it, allowing the developer the freedom to visualize the survey answers however wished to.

3.4.2.2 Statistics Provider

Now that we know how the front-end acts when needing to visualize the questions, we should demystify who provides the front-end of the statistics. The statistics for the term, stats or other kinds of questions are provided by the back-end. Because the back-end handles all the data store and data processing, I have decided to call the part of the back-end that is responsible for the statistics as 'Statistics Provider'. The whole backend will be interpreted in technical analysis. For the time being, it is important to understand that the whole platform communicates through REST endpoints, with a client-server architecture. There is a specific endpoint in our backend, which is responsible for providing the caller with the statistics for any question that was requested. It requires the question id, type and the survey that we are interested in and provides the stats in the format that we have already discussed, in Results Rendering. Let us visualize the process and then follow up with the code.

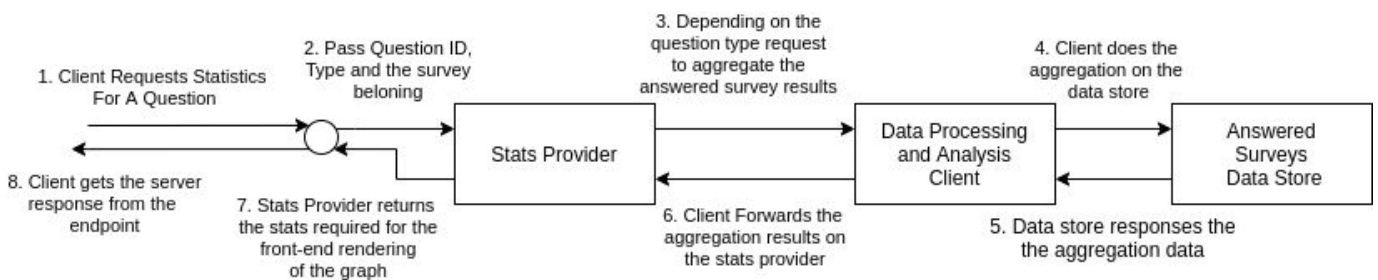


Diagram 10: Results Reporting Statistics Provider

Each one of the above steps, along with their design and technical details will be clarified in the technical analysis. What is important here is to make a firm introduction to the architecture of our service. More about the Data Processing and Analysis Client and our Data store will be described in the following chapter, that of Visualization Wizard. Our focus here is the Stats provider. It is the intermediate between the front-end and the data

store and analysis. The Data Processing and Analysis client queries the data store however it is ordered by the Stats Provider. So, let us shift to the Stats Provider inner logic, starting by its code:

```

getQuestionStats = async (req, res) => {
  console.log("→ [getQuestionStats] controller will handle the request");
  const survey = req.query.survey;
  const questionID = req.query.questionId;
  const questionType = req.query.questionType;
  var answers_field;
  switch (questionType) {
    case "Number":
      answers_field = `answers.${questionID}`;
      elastic
        .statsAggregation(indexName, survey, answers_field)
        .then((result) => {
          let response_data = {
            total_answers: result.hits.total.value,
            stats: result.aggregations.aggs_result,
          };
          res.send(response_data);
        });
      break;
    default:
      answers_field = `answers.${questionID}.keyword`;
      elastic
        .termsAggregation(indexName, survey, answers_field)
        .then((result) => {
          let response_data = {
            total_answers: result.hits.total.value,
            stats: result.aggregations.aggs_result.buckets,
          };
          res.send(response_data);
        });
  }
};

```

Code 15: Results Reporting Statistics Provider

The Stats provider contains once again the same conditional logic of the Question and Visualization Rendering. Based on the question type, asks the client that handles the data storing to do a stats or terms aggregation. Then, gets the responses and packs up the statistics for the front-end rendering to occur, in the format that we showed in the Results Rendering section.

3.4.2.3 Summing up

To sum up the whole Results Rendering Process. Throughout the whole code and design process the number one priority was to keep a tight and strict separation of concerns. The front-end asks for the question stats. The backend manages its systems to return those stats. Then the front-end decides to render them. Let us do a last visualization of the whole process and more details will follow in the rest of this thesis.

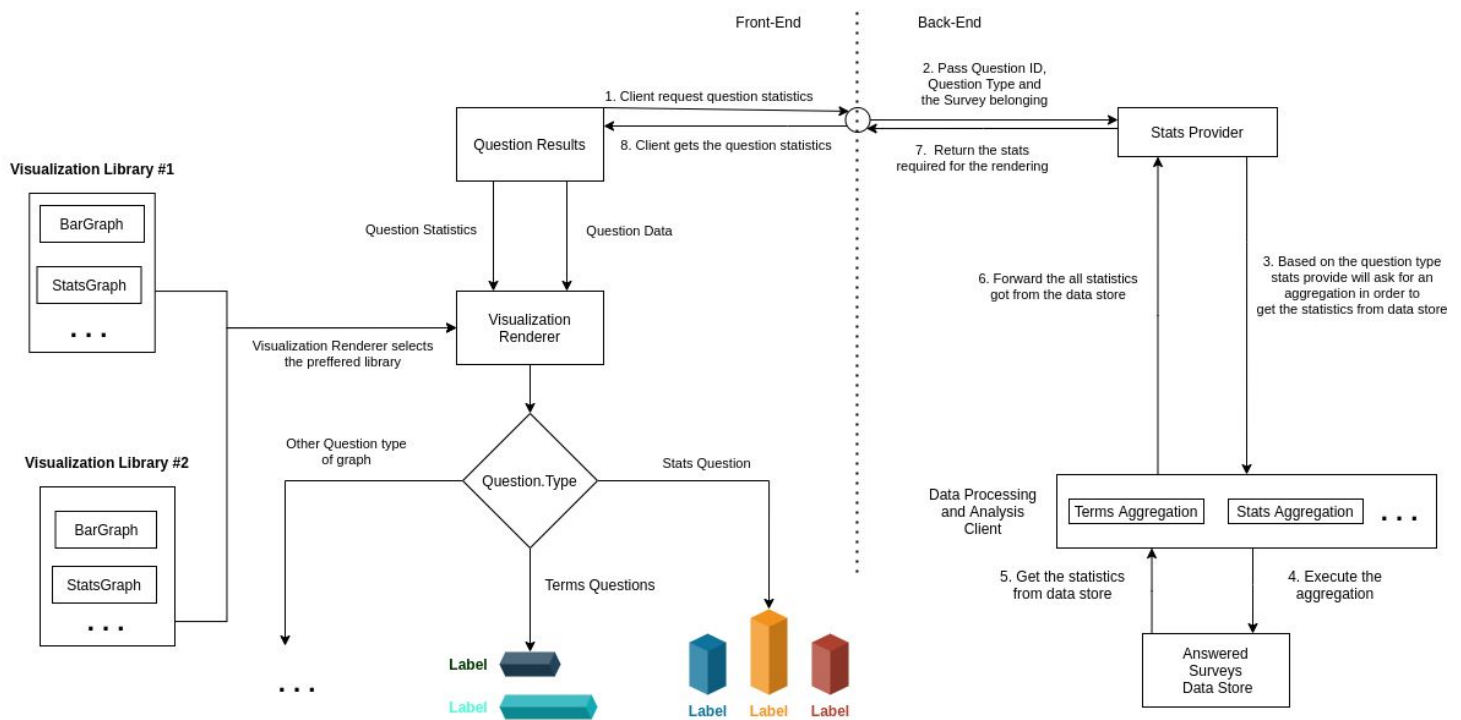


Diagram 11: Results Reporting Complete Process

The complete visualization of how the question's answers are getting graphed on the user screen. We are now ready to move on to the Visualization Wizard.

3.4.3 Visualization Wizard

It is now time to proceed to the explanation of the last feature which our survey tool offers. The Visualization Wizard. Given a number of answered surveys, it's of course important to have a way to visualize the answers to specific questions. But it would be great if we could combine and aggregate different questions in different orders and contrasting fields. Our platform offers a way to experiment with the results, allowing the user to explore and elicit their own deductions. The whole process of the Question aggregation and visualization is called Visualization Wizard, because it abstracts away from the user a lot of complexity and it offers a wide variety of options to choose and 'play' with. The Visualization Wizard was the most difficult part of this thesis. It required a deepening on the technology stack used, designing many discrete and unlike systems, form multiple layers of communication and applying different programming techniques. In the Visualization Wizard there will be a comprehensive introduction to the technical analysis of the implementation, in order to understand the building process. The whole process will be explained step by step, so that it can be fully grasped and interpreted.

3.4.3.1 General

Before advancing to the design and technical details of the implementation, for once more, let us showcase how the Visualization Wizard works from the user's perspective. The user gets an UI, in which they can choose which questions to combine, in the desirable order. Then they can adjust and tune the different parameters and filters that each question offers for the data analysis to happen. Different question types offer different types of parameterization .

After deciding which questions will aggregate, along with their order and parameters, they can send the request to get the results. The server will respond with the results, given in a nested format, depending on the question order. This will be ascertained soon but it is important to clarify the logic with which the whole system processes the questions.

Having different types of questions, and with each one having different possible answers, there should be a way to represent the results. For each question that the user decided upon, the code will analyze the answers and split them into different buckets, with each bucket containing the answered surveys with the specific answer to the question specified. Then for each bucket, continue to aggregate with the following questions, resulting in a set of buckets with nested buckets. Then it calculates the last statistics based on the last

question. Sounds confusing? All the ‘how’ details will be described in the following chapters. For the time being, let us focus on what the Wizard does.

Let us start off by giving 3 examples. We will use the Visualization Wizard for the Students survey that we have created, which concerns their fears about the transition to the workforce and how the university contributed in that direction.

- 1) I will begin with the simplest possible example. Let us ask the wizard to extract the students who are ready for the workforce, or not, based on their gender.

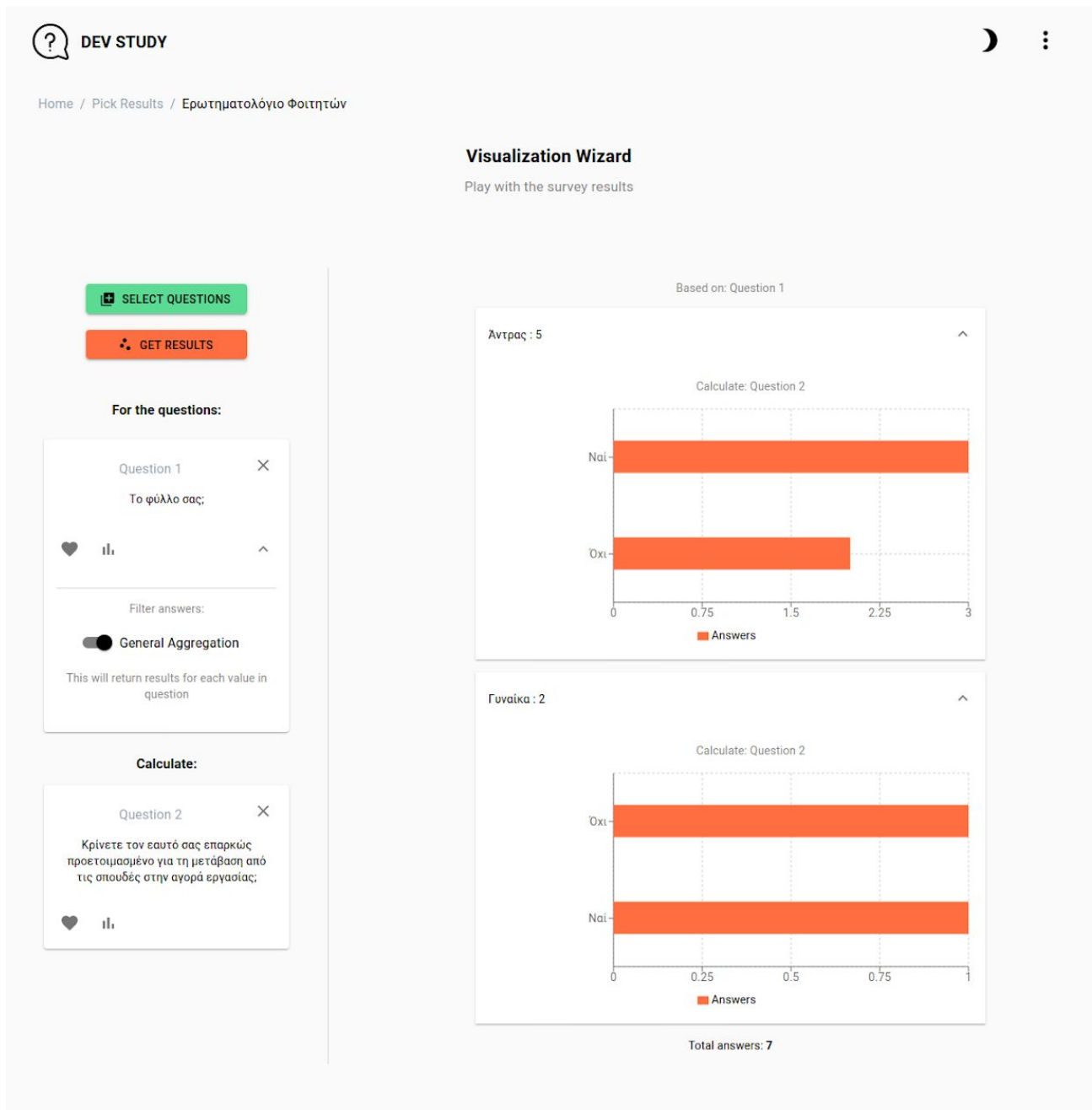


Figure 26: Visualization Wizard Example 1

In the above example, the wizard based on the gender question, will split the answered surveys to those of the participants which answered Male and to those which answered Female. Then for each set of answered surveys, it will retrieve the statistics for the people that are ready for the workforce and those not ready for it, and visualizes it. But what if we wanted only the male participants ? Let us see how this can be done:

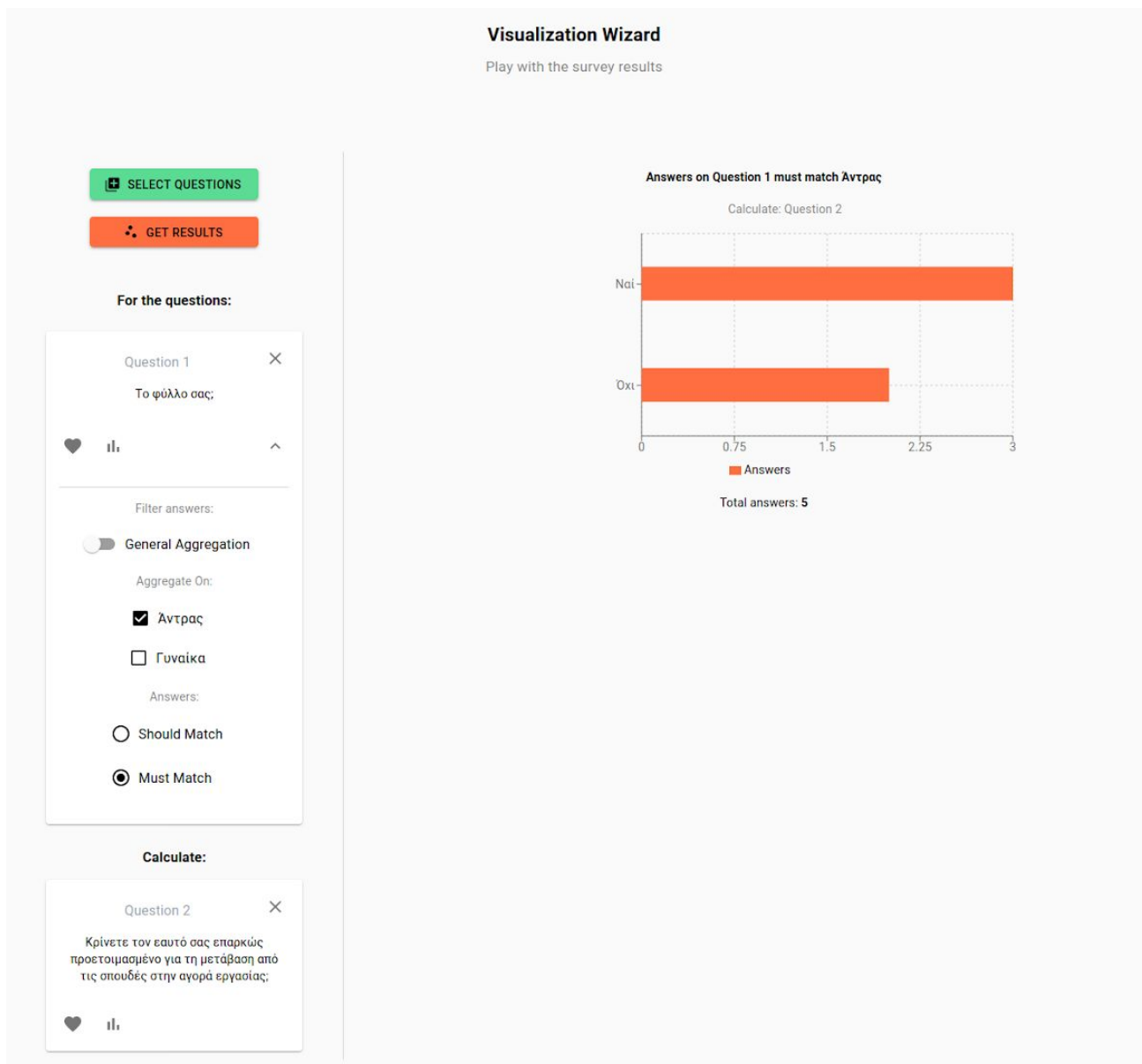


Figure 27: Visualization Wizard Example 1 - Parameterized

Question 1 is no longer a General Aggregation, which produces results for all answers in the survey, but rather more specialized. Here we only target the male participants, so we choose only that Answer and select the Must match parameter. (We will explain all the possible parameters in the design phase).

- 2) Let us try that on a number of types of questions. I will ask the Wizard to extract the participants that rated their well being (diet, sleep quality etc) above average and rate the university for its contribution to their personal growth.

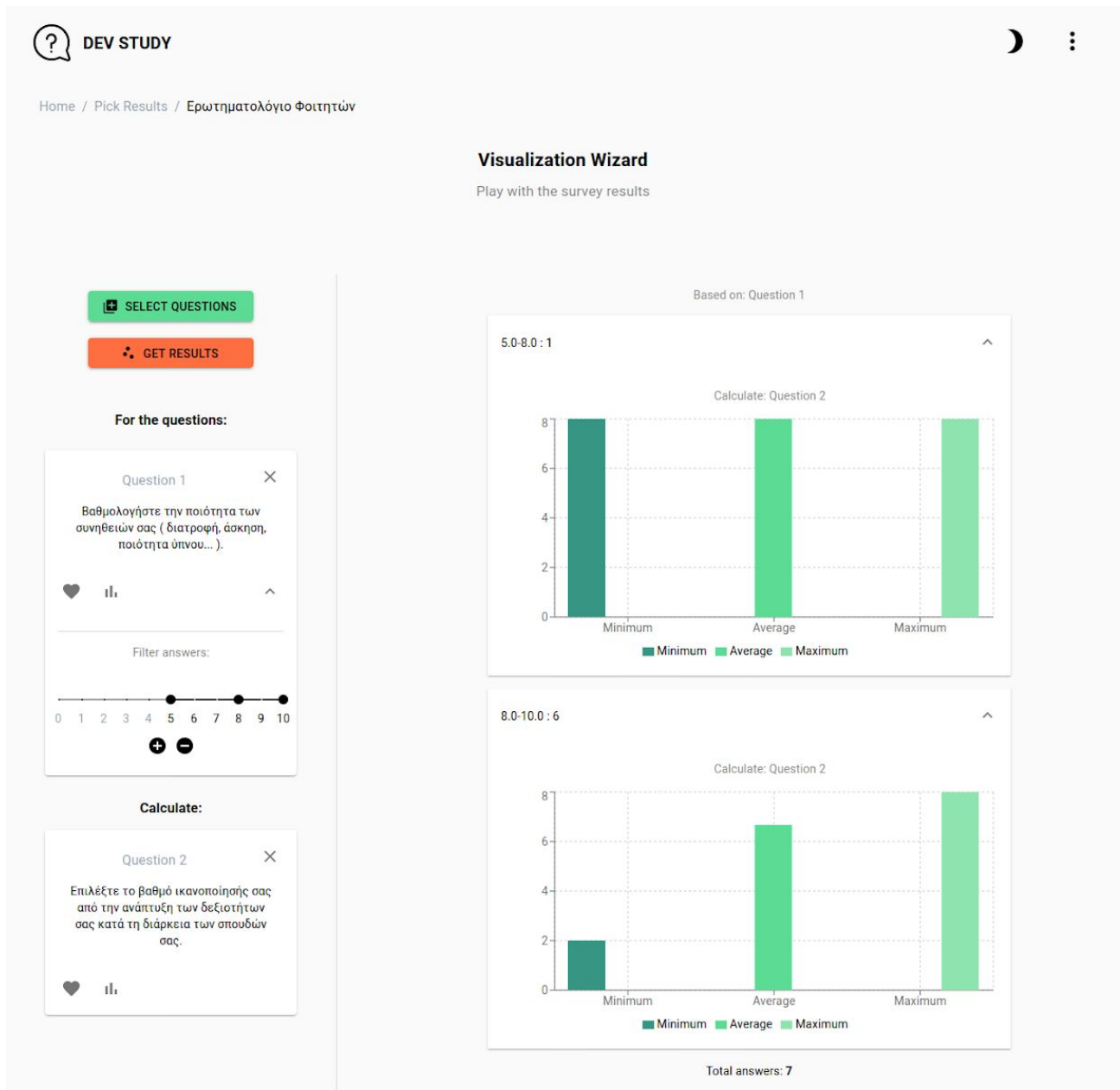


Figure 28: Visualization Wizard Example 2

Observe now that the filters are not tied to a set of answers but rather to the scale of the numeric possible answers. Individuals can specify the range of answers they wish to aggregate data on and then select the question on which we do the calculations on , either that being a stats question or a terms question.

- 3) For the final step, let us try aggregating 3 questions. The first question that we want to aggregate is the students motives for pursuing computer science, then split them up based on their gender and project the above answered survey sets to whether those people consider themselves ready for the workforce.

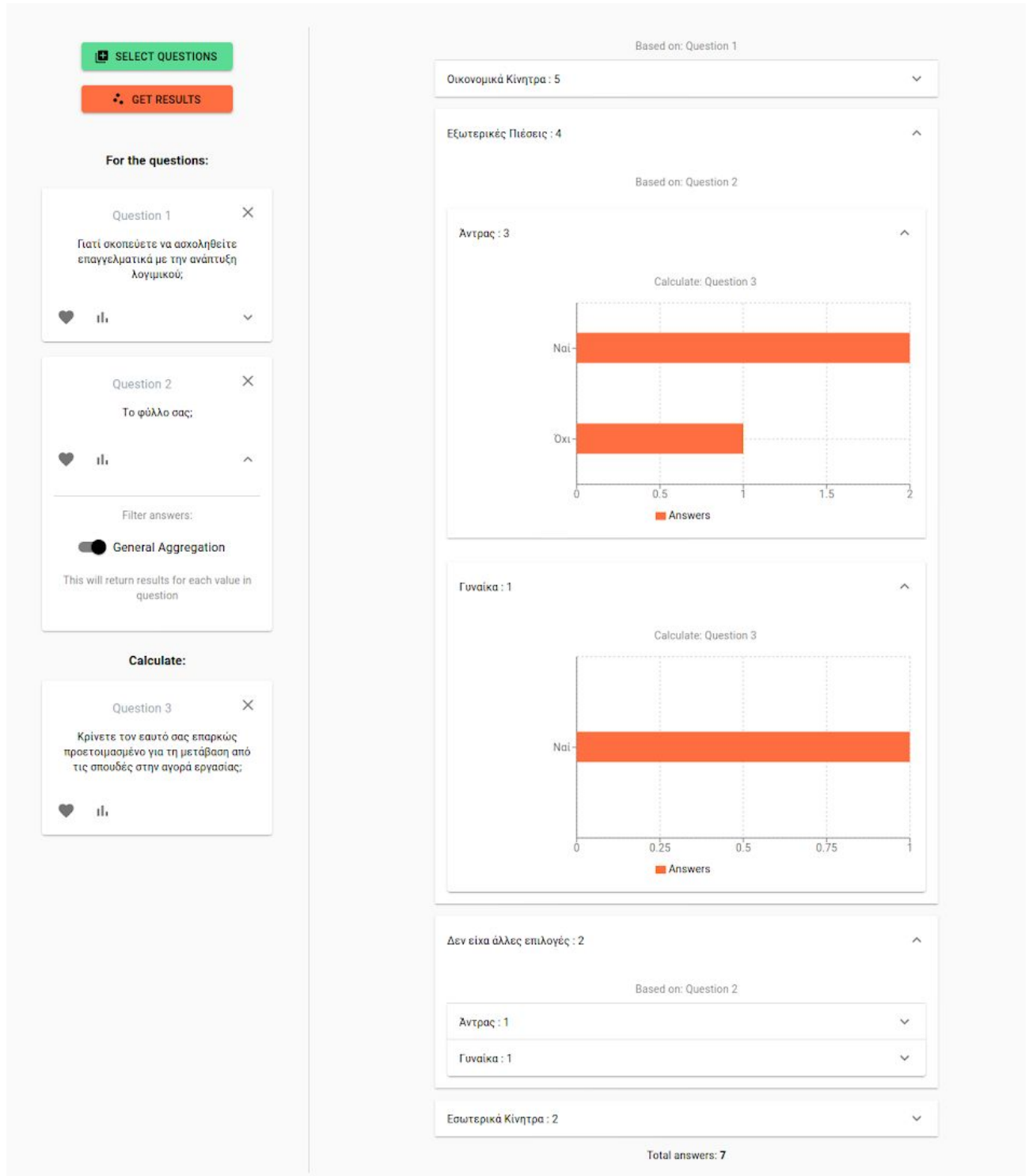


Figure 29: Visualization Wizard Example 3

The Visualization Wizard now splits the answers based on the participants motives. Then for each motive divides answered surveys based on the gender and last does the calculation by rendering the question statistics provided.

Of course the list of examples can be infinite. People that would like to explore the survey results, can select all the possible combinations, and an arbitrary number of questions and get their results. Of course the visualization wizard tool does not guarantee any connection between the data and answers. It only displays the answers based on user preferences. Fundamentally, each question answered is just a counter. It is up to the individual's discretion to extract results that correspond to reality.

And now that we have cleared up what the Visualization Wizard does, let us move on to **How** it does it. Once again we incept with the design decisions and progress with the technical details.

3.4.3.2 Analysis

There are a lot of questions emerging about the implementation. How do we describe all those questions and their parameters in a way that the data store can process? What does the data store respond to? How does the visualization render the combined results? All those questions concern separate entities in the code implementation. And these entities communicate in different ways in a specific sequence. Let us outline the whole process and get into detail about each entity next.

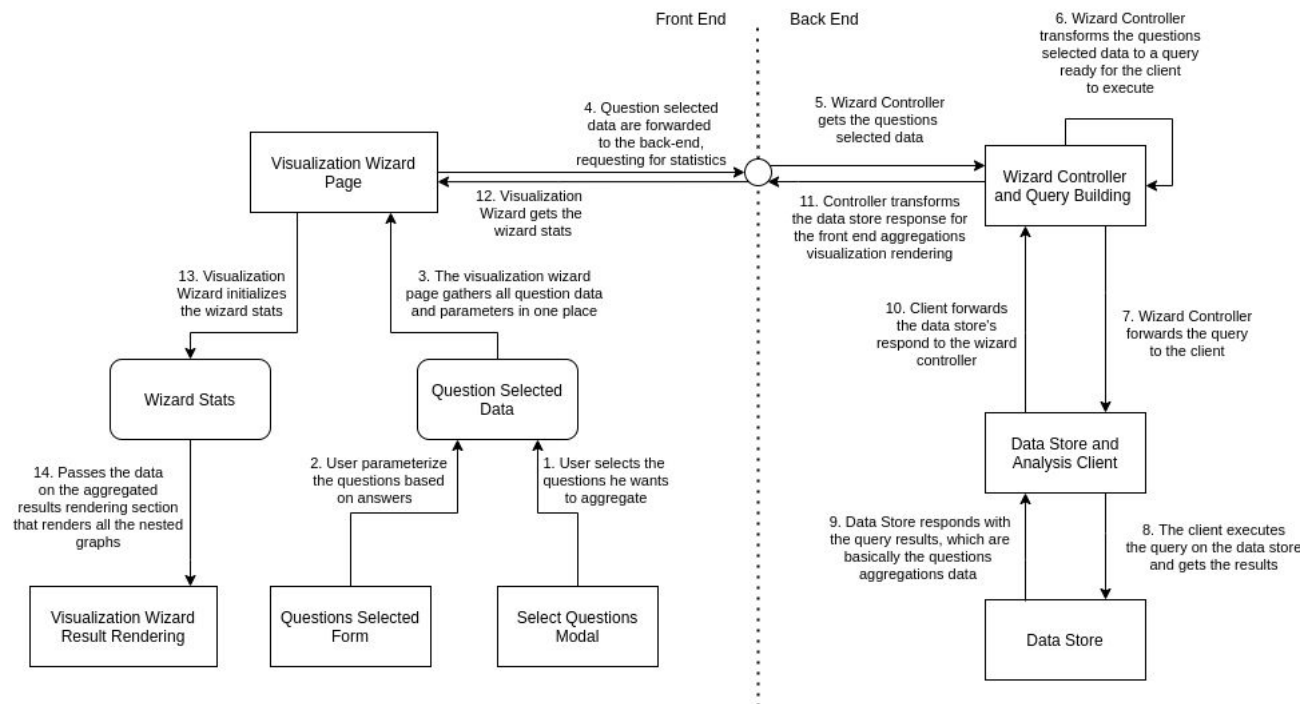


Diagram 12: Visualization Wizard Process Abstract

The take-away from this outline is to understand that the front-end summarizes the question data and their parameters into a central place. Then passes it on to the back-end. The back-end controllers get those questions selected data, apply some levels of transformation and build the query for the data store. The client executes the query and gets the results passing them to the Wizard controller. The wizard gets the data store response which contains all the information about the aggregations occurred and packs up the crucial information into a json file that the front-end requires for rendering. All the wizard statistics that the wizard controller just created, are passed from the visualization wizard page to the Rendering process that paints the nested graphs based on each question, on the client.

Let us now start analyzing each entity, while presenting the whole process step by step.

I. Statistics Provider

Unlike the Results Reporting, for the Visualization Wizard documentation we will start off with the Statistics Provider of the Visualization Wizard and then we will resume with the Wizard Results rendering process.

The Statics Provider is part of the whole outline:

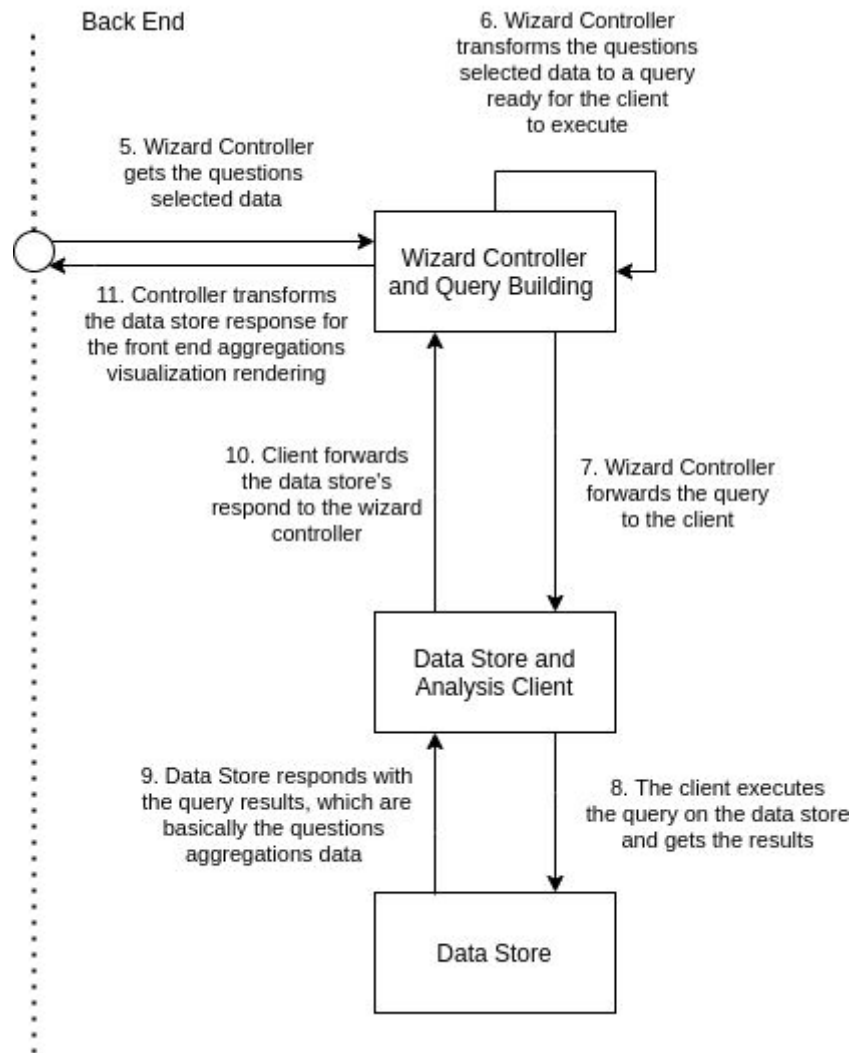


Diagram 13: Visualization Wizard Statistics Provider Abstract

The Statistics Provider job for the Visualization Wizard tool is to provide the front-end of the statistics required for it to render the aggregated results. The whole process makes a full-circle with a set of intermediate 'stations' where the data format changes. The transformations states and flow is shown in the diagram below. In this introduction phase we will only put the 4 basic data states and by the end of this chapter we will preview the whole process.

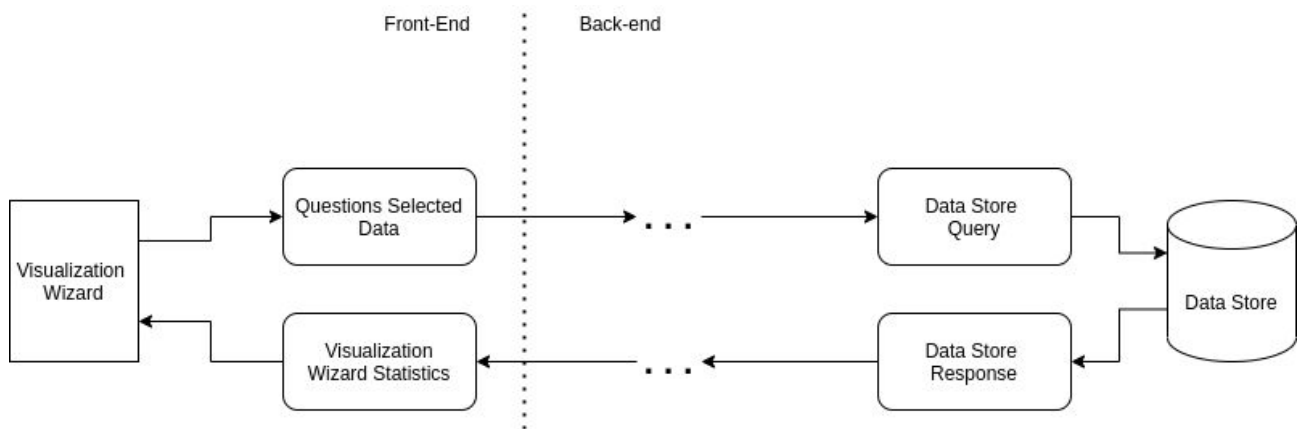


Diagram 13: Data Transformation Process Abstract

The data starts off with the Questions Selected Data format, after a set of transformations, it is converted to a data store query that will be executed on the data store. Then the statistics provider will get the data store response and after applying a transformation process, it will provide the Visualization Wizard with the wizard statistics needed , in order to visualize the results.

i) Questions Selected Data to Query process

The data-flow starts from the Questions Selected Data format and is converted to the Query for the datastore. The forward data flow will be showcased, and then the backwards will follow.

A. Questions Selected Data

Before explaining the Questions Selected Data format, let us see where we are:

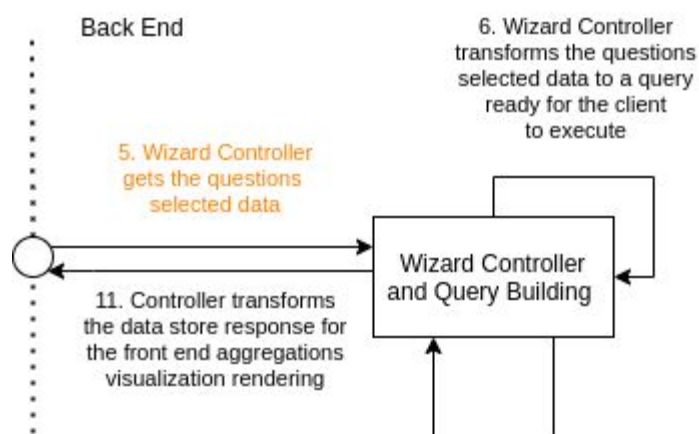


Diagram 15: Visualization Wizard Statistics Provider - Phase 1

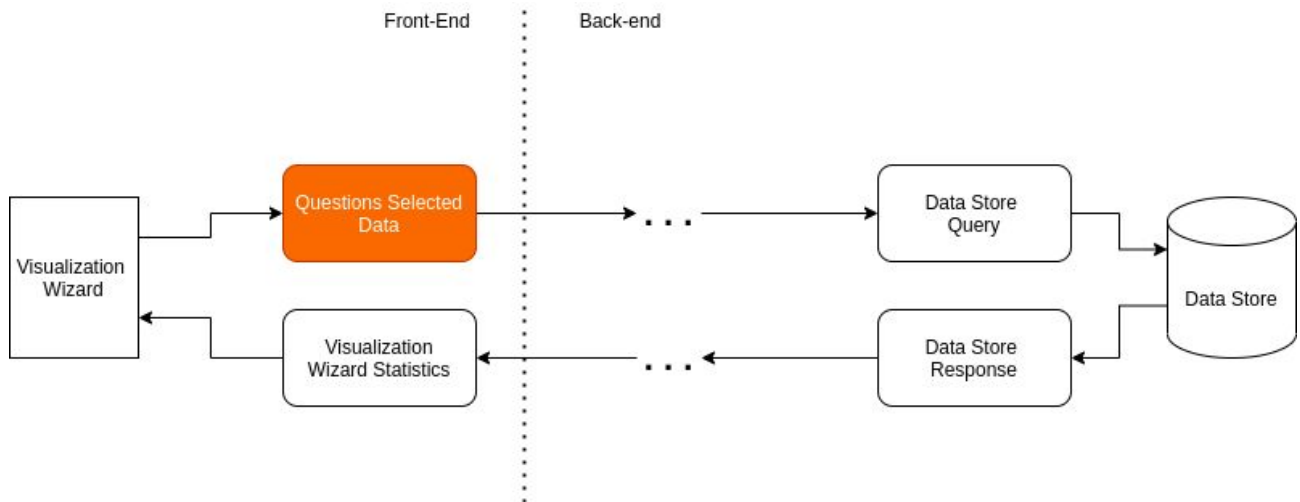


Diagram 16: Data Transformation Process - Phase 1

We are at the beginning of the Statistics Providing process, before we head to the wizard controller that converts the Questions Selected Data.

All the questions data, along with their parameters and filters applied, should be described in a data format, special for the back-end controllers to understand and process. The Questions Selected Data format looks like the following (We will use the first example of the Visualization Wizard).

```
QuestionsSelectedData : {
  q_id: <Q_ID>,
  q_type: <Q_Type>',
  q_args: <Q_Args for the Specific type of question>
},...
]
```

Where q_args for Terms kind of questions look like this:

```
q_args: {
  aggregateOn: [],
  aggregateOp: 'must',
  isAggrGeneral: true
}
```

and q_args for Stats kind of questions look like this:

```
q_args: {
  range: [ 0, 5, 10 ]
}
```

Figure 30: Question Selected Data Format

The front-end needs to create the above data format and give it to the back-end controllers, in order to retrieve the wizard stats data about the aggregations on those questions. For each question that the user selects, based on the question type, we have different kinds of parameters and filters, so that they can have meaning. We retain the same logic in the Visualization wizard, that of different kinds of questions when it comes to visualization and aggregation. Each question data which belongs to the array of the questions selected that will pass to the back-end controller, contains the basic information like the question's id and type and the questions arguments that the user has specified. The Stats Questions need the range of answers we want to aggregate answers to. For the Terms Questions we can apply a general aggregation, if we want to aggregate results based on all possible answers, or non-general, meaning that the user should select the answers they want to apply the aggregation on along with their in between relationship. Should means that the possible answers should match, while the Must operations means that the answers Must match.

B. DataStore Query

We will skip the phase of the transformation process and move on the end result, the Query. It is wise to first understand how the end query looks like and how the initial data information was, so that the whole transformation process can be interpreted. Let us see where we are:

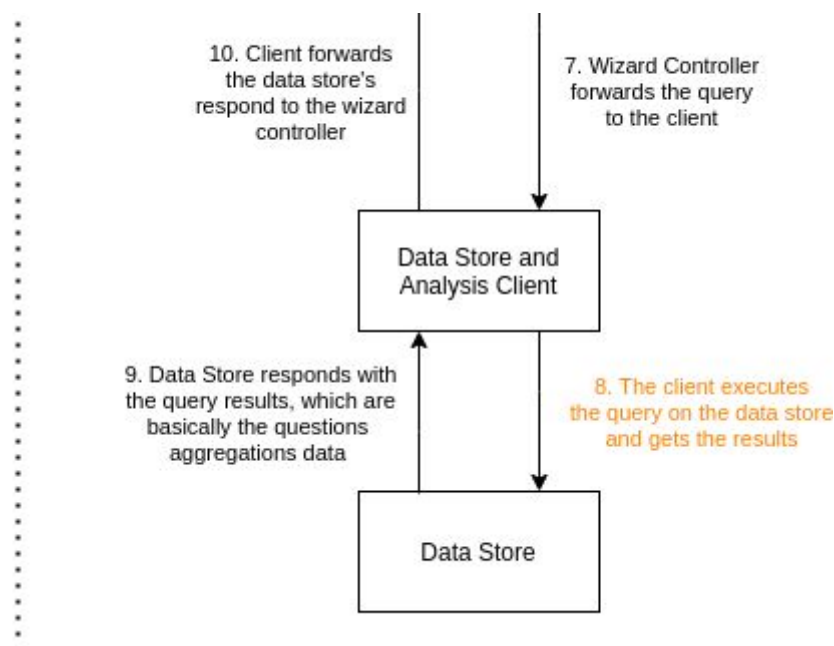


Diagram 17: Visualization Wizard Statistics Provider - Phase 2

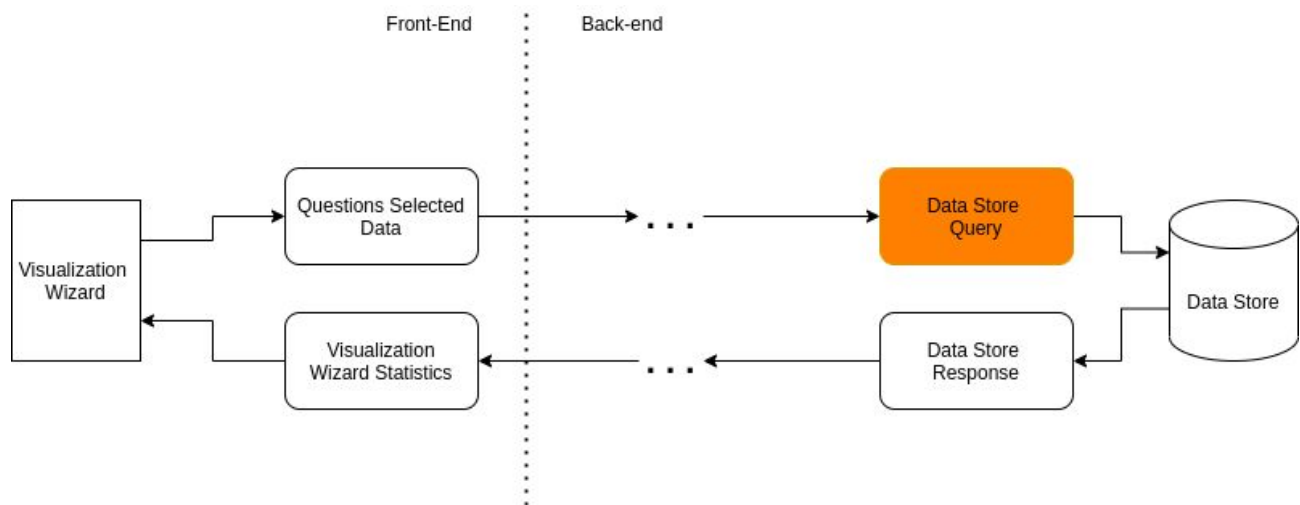


Diagram 18: Data Transformation Process - Phase 2

The query follows the format the data store indicates, so that it can be executed and yield the aggregation results. Let us see how the end-query looks like, for two different examples:

- *Example 1:*

We will ask the Visualization Wizard to extract the information regarding :

- The participants who answered “Innate motives” or “No other choices” to the question about computer science motives .
- For the question whether they are satisfied from the university a general aggregation, meaning that we want to analyze the end results based on all the possible answers
- And the question that we want to calculate the answers to, is to rate their well being.

This question sequence is a Non-general Terms Question, a General Terms Question and Stats Question.

The query that our data store system will execute is:

```

endQuery:
{
  size: 0,
  query: {
    bool: {
      must: [ { match: { id: 'STUD' } } ],
      should: [
        { match: { 'answers.Q01.keyword': 'Εσωτερικά Κίνητρα' } },
        { match: { 'answers.Q01.keyword': 'Δεν είχα άλλες επιλογές' } }
      ]
    },
    aggs: {
      agg_0: {
        terms: { field: 'answers.Q06.keyword' },
        aggs: { agg_1: { stats: { field: 'answers.Q29' } } }
      }
    }
  }
}

```

Figure 31: DataStore Query Format - Example 1

Let us take this step by step. The logic behind the data store system is actually Elasticsearch's technology, that we will see in the Technical Analysis.

- The *size* argument indicates how many answered surveys should the data store bring. We only want the statistics, so we give zero value to the argument.
- The next attribute is the query which is basically the parameters that we specify for the aggregation to happen. The *bool* key contains the attributes the answered surveys should satisfy. The *must* object contains all the criteria that the answered surveys must match. As seen on the example, it must match the survey id, meaning that we want all the answered surveys for a specific survey. The *should* object contains all the answers that the answers should match. In our example, we want the participants that answered “Innate Motives” or “No other choices”.
- The last property is the *aggs* property, which is basically all the aggregations that should be applied to the answered surveys. To all the General Aggregations we want to apply a terms aggregation, meaning that we split all the answered surveys to different sets based on each question's answer. In the survey management we nest all the different aggregations, concluding in the last one, which is basically the question we calculate our results to. In this case, it is just a stats aggregation, that the backend knows how to handle.

- *Example 2:*

Let us see another simpler example, so as to show a different type of aggregation that happens on our data-store. Now we will see the participants that:

- Rate their well-being above average, defining the ranges from 5 to 6, from 6 to 8 and from 9 to 10.
- Calculate the above answered surveys based on the question about their social life.

Let us see what the data-store needs for Stats kind of questions

```
endQuery: {
  size: 0,
  query: {
    bool: {
      must: [ { match: { id: 'STUD' } } ],
      should: []
    }
  },
  aggs: {
    agg_0: {
      range: {
        field: 'answers.Q29',
        ranges: [ { from: 5, to: 6 }, { from: 6, to: 8 }, { from: 10, to: 9 } ]
      },
      aggs: { agg_1: { stats: { field: 'answers.Q30' } } }
    }
  }
}
```

Figure 32: DataStore Query Format - Example 2

Here the stats question will be analyzed as a stats aggregation which the data store will handle, and only requires the different ranges that the answered surveys will be split into, based on the question answer. The question that we calculate our data on is a Number question, so the last aggregation will be a stats aggregation, as well .

The underlying data store mechanisms, like how it provides the results, how it handles the answered surveys etc, depends on the technology stack used. In our services we use elasticsearch as the data store and it will be explained firmly in the technical analysis chapter.

C. Transformation Process

Now that the initial and final form of the question selected data has been clarified, let us focus on the transformation process that happens in the wizard controller.

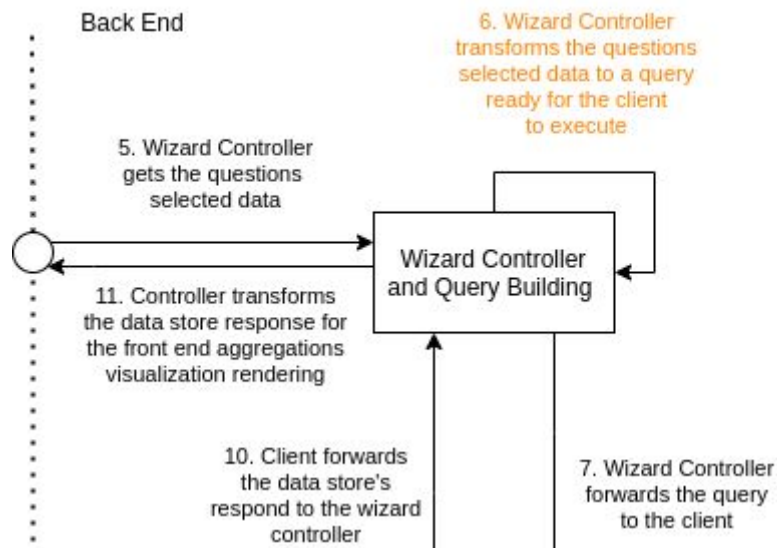


Diagram 19: Visualization Wizard Statistics Provider - Phase 3

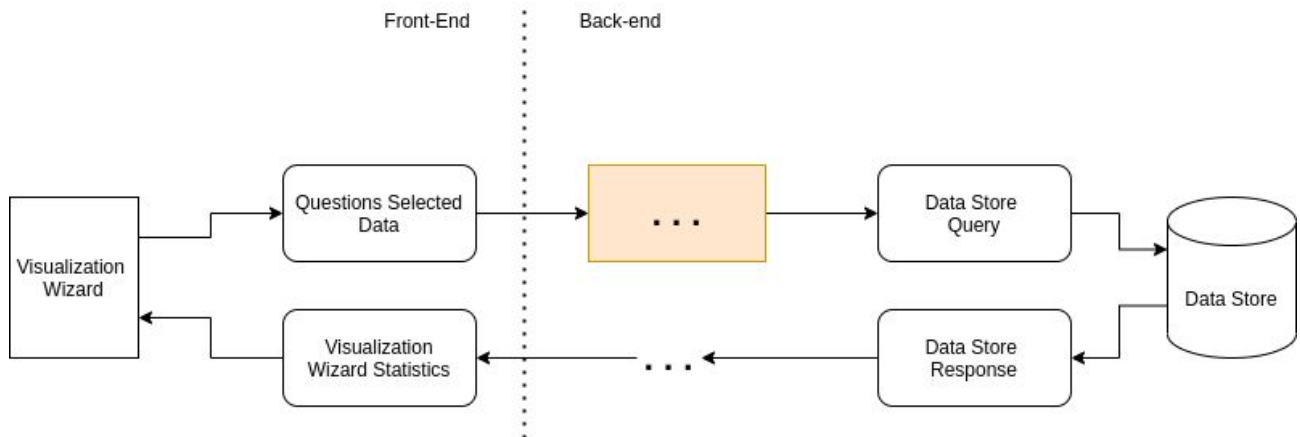


Diagram 20: Data Transformation Process - Phase 3

The transformation phase from the Questions Selected Data to the Data Store Query has **3 major phases**. Let us see them:

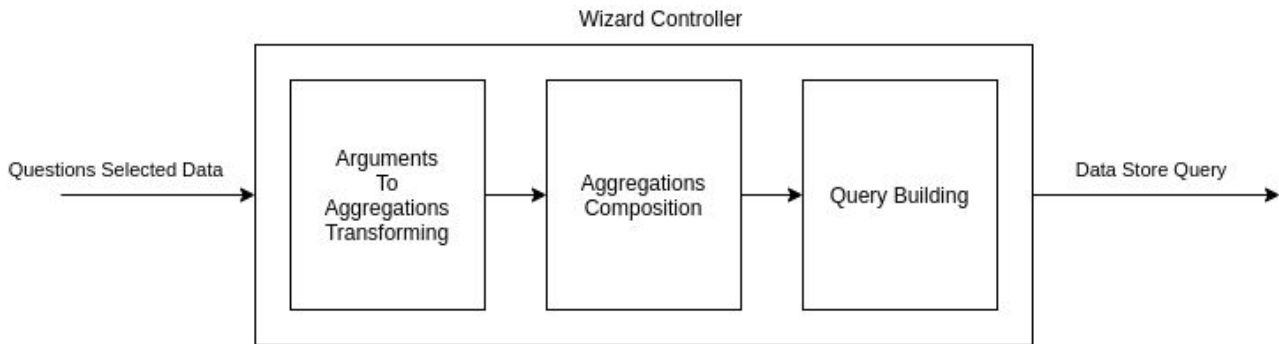


Diagram 21: Forward Transformation Process Abstract

a) Arguments To Aggregations Transforming

Imagine having the following data for the questions selected from the user:

A. Received the following question parameters for the aggregations

```
[
  {
    q_id: 'Q01',
    q_type: 'SetOfStrings',
    q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
  },
  {
    q_id: 'Q02',
    q_type: 'String',
    q_args: {
      aggregateOn: [ 'Naí', '0x1' ],
      aggregateOp: 'should',
      isAggrGeneral: false
    }
  },
  { q_id: 'Q15', q_type: 'Number', q_args: { range: [ 0, 5, 10 ] } },
  { q_id: 'Q30', q_type: 'Number', q_args: { range: [ 0, 5, 10 ] } }
]
```

As we have already seen, we need to define different kinds of filters and aggregations for the Query to be executed. That 3 basic components that the Query needs are the *shoulds*, *musts* and the *aggs*. So, we must translate, we must retrieve the questions arguments one by one and translate them to these basic components. The general aggregation questions are translated to terms aggregations, the non-general aggregations are translated to either *musts* or *shoulds* based on the aggregation operation and the stats type questions are converted to a stats aggregation.

After the arguments transformation process, the different aggregations and filters

are gathered to array data structures for the Aggregations Composition to happen:

```

B. Processed the question parameters and digested them to:

Converted the q_args to aggregations:

[
  { agg_0: { terms: { field: 'answers.Q01.keyword' } } },
  {
    agg_1: {
      range: {
        field: 'answers.Q15',
        ranges: [ { from: 0, to: 5 }, { from: 5, to: 10 } ]
      }
    }
  },
  { agg_2: { stats: { field: 'answers.Q30' } } }
]

Musts:

[ { match: { id: 'STUD' } } ]

Shoulds:

[
  { match: { 'answers.Q02.keyword': 'Ναι' } },
  { match: { 'answers.Q02.keyword': 'Όχι' } }
]

```

Figure 33: Digested Questions Selected Data

Each of the Query different components are firstly gathered two these three distinct data structures. This happens so that we can be flexible with the Aggregations Composition process. The code for the arguments to aggregations transforming is the following:

Code 16: Arguments To Aggregations Transforming

```

////////// 2. Digest the question aggregation parameters to the following d
let must = [{ match: { id: surveyId } }];
let should = [];
let aggs = [];
var new_agg;
let agg_indx = 0;
// For each question, fill the different aggr props
q_params.forEach((question, index) => {
  let isFinal = index + 1 === q_params.length;
  if (!isFinal) {
    // If it is not the last
    switch (question.q_type) {
      case "Number":
        new_agg = {
          [`agg_${agg_indx}`]: {
            range: {
              field: `answers.${question.q_id}`,
              ranges: (function (ranges) {
                let ranges_processed = [
                  { from: ranges[0], to: ranges[1] },
                ];
                for (var i = 1; i < ranges.length; i++) {
                  if (i % 2 === 0) {
                    ranges_processed.push({
                      from: ranges[i - 1],
                      to: ranges[i],
                    });
                  }
                }
                return ranges_processed;
              })(question.q_args.range),
            },
          },
        };
        aggs.push(new_agg);
        agg_indx++;
        break;
      default:
        if (question.q_args.isAggrGeneral) {
          new_agg = {
            [`agg_${agg_indx}`]: {
              terms: {
                field: `answers.${question.q_id}.keyword`,
              },
            },
          };
          aggs.push(new_agg);
          agg_indx++;
        } else {
          switch (question.q_args.aggregateOp) {
            case "must":
              question.q_args.aggregateOn.forEach((value) => {
                let new_must = {
                  match: {
                    [`answers.${question.q_id}.keyword`]: value,
                  },
                };
                must.push(new_must);
              });
              break;
            default:
              question.q_args.aggregateOn.forEach((value) => {
                let new_should = {
                  match: {
                    [`answers.${question.q_id}.keyword`]: value,
                  },
                };
                should.push(new_should);
              });
            }
          }
        }
      }
    }
  } else {
    // The last question, is the question we do the last aggr. on
    switch (question.q_type) {
      case "Number":
        new_agg = {
          [`agg_${agg_indx}`]: {
            stats: {
              field: `answers.${question.q_id}`,
            },
          },
        };
        aggs.push(new_agg);
        break;
      default:
        new_agg = {
          [`agg_${agg_indx}`]: {
            terms: {
              field: `answers.${question.q_id}.keyword`,
            },
          },
        };
        aggs.push(new_agg);
      }
    }
  }
});

```


Briefly, the above code interprets the Questions Selected Data format and is based in a sequence of condition logic, it manages to produce the different components of the Query.

Now, let us see how the Aggregations Composition merges the aggregations and why it does that.

b) Aggregations Composition

In this phase all the 3 core components of the query in array-like structures are flexible to act however we choose. The *must* and *should* aggregations are ready for the Query. But the Aggregations are a bit more complex. Elasticsearch requires the *aggs* to be a nested object, so that it can recursively apply the aggregations in the buckets created which are created every time answers split based on the specifications of the user. So this phase gets the aggregations that gathered and nests them in one nested aggregation format. Let us see the aggregation result after the composition of the different aggregations.

Before the aggregations composition:

```
[
  { agg_0: { terms: { field: 'answers.Q01.keyword' } } },
  {
    agg_1: {
      range: {
        field: 'answers.Q15',
        ranges: [ { from: 0, to: 5 }, { from: 5, to: 10 } ]
      }
    }
  },
  { agg_2: { stats: { field: 'answers.Q30' } } }
]
```

After the aggregations composition:

```
{
  agg_0: {
    terms: { field: 'answers.Q01.keyword' },
    aggs: {
      agg_1: {
        range: {
          field: 'answers.Q15',
          ranges: [ { from: 0, to: 5 }, { from: 5, to: 10 } ]
        },
        aggs: { agg_2: { stats: { field: 'answers.Q30' } } }
      }
    }
  }
}
```

Figure 34: Before and After Aggregations Composition

With that successfully finished, the query building can easily be done. Before proceeding to the final step, let us see how the code does the aggregations composition:

```
// This functions get object's referce by it's path within the object
Object.byString = function (o, s) {
  s = s.replace(/\[(\w+)\]/g, ".$1"); // convert indexes to properties
  s = s.replace(/^\./, ""); // strip a leading dot
  var a = s.split(".");
  for (var i = 0, n = a.length; i < n; ++i) {
    var k = a[i];
    if (k in o) {
      o = o[k];
    } else {
      return;
    }
  }
  return o;
};

let aggs_processed = aggs[0];
let aggPosPath = "agg_0"; // this is the path of the aggregation that will nest the new query
for (var i = 1; i < aggs.length; i++) {
  // get the reference of the object that will nest the query
  let ref = Object.byString(aggs_processed, aggPosPath);
  // get the new aggr in order to nest it to the aggr already created
  let new_agg = { aggs: aggs[i] };
  Object.assign(ref, new_agg); // add the new aggregation to the aggregation already created
  // create the path of the created aggr, in order to store the next one
  aggPosPath += ".aggs.agg_" + i;
}
```

Code 17: Aggregations Composition

c) Query Building

So far, we have successfully managed to gather the three components for the Query Building. Let us see them all together:

```
Aggs:
{
  agg_0: {
    terms: { field: 'answers.Q01.keyword' },
    aggs: {
      agg_1: {
        range: {
          field: 'answers.Q15',
          ranges: [ { from: 0, to: 5 }, { from: 5, to: 10 } ]
        },
        aggs: { agg_2: { stats: { field: 'answers.Q30' } } }
      }
    }
  }
}

Musts:
[ { match: { id: 'STUD' } } ]

Shoulds:
[
  { match: { 'answers.Q02.keyword': 'Noi' } },
  { match: { 'answers.Q02.keyword': 'Oxl' } }
]
```

Figure 35: Query Building Components

The Query Building happens by creating the Elasticsearch's query data format and just appending each of the components in the right position. Let us understand this by the code example:

```

////////// 3. Build the query for the elastic client
let elasticsearch_query = {
  size: 0,
  query: {
    bool: {
      must: must,
      should: should,
    },
  },
  aggs: aggs_processed,
};

```

Code 18: Data Store Query Building

This last command is just a variable that contains the final query. The end-query is finally ready. Let us preview the final result before executing it on the Data Store.

C. Built the following elasticsearch query:

```

{
  size: 0,
  query: {
    bool: {
      must: [ { match: { id: 'STUD' } } ],
      should: [
        { match: { 'answers.Q02.keyword': 'Noi' } },
        { match: { 'answers.Q02.keyword': 'Oxi' } }
      ]
    }
  },
  aggs: {
    agg_0: {
      terms: { field: 'answers.Q01.keyword' },
      aggs: {
        agg_1: {
          range: {
            field: 'answers.Q15',
            ranges: [ { from: 0, to: 5 }, { from: 5, to: 10 } ]
          },
          aggs: { agg_2: { stats: { field: 'answers.Q30' } } }
        }
      }
    }
  }
}

```

Figure 36: DataStore Query Built

The forward data flow has ended. Let us summarize the whole Questions Selected Data which the front-end provides to the Data Store Query process with the following diagram:

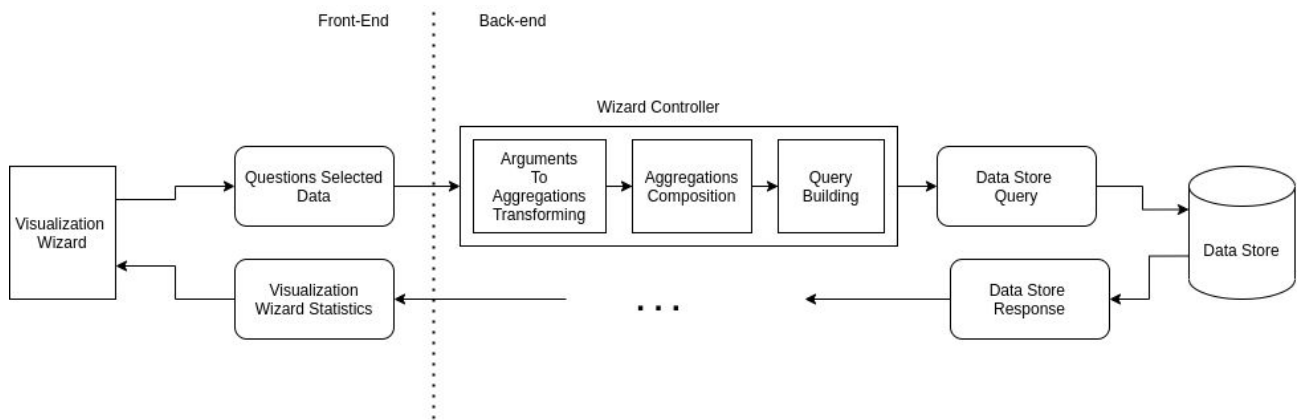


Diagram 22: Data Transformation Process - Forward Complete

It is time for us to delineate the backward data flow. The conversion from Data Store Response to the Wizard Stats, which the Visualization Wizard Page will accumulate to render the statistics.

ii) Aggregations Results to Wizard Statistics process

The process of the Wizard Stats providing to the front-end from the Data Response that the data store got us, is much more simpler. Let's analyze it, starting for the back-end response:

D. Aggregations Results

The data store responds and yields a reply with the following format.

D. Received the query results from the elasticsearch:

```
{
  took: 2,
  timed_out: false,
  _shards: { total: 1, successful: 1, skipped: 0, failed: 0 },
  hits: { total: { value: 7, relation: 'eq' }, max_score: null, hits: [] },
  aggregations: {
    agg_0: {
      doc_count_error_upper_bound: 0,
      sum_other_doc_count: 0,
      buckets: [
        {
          key: 'Οικονομικά Κίνητρα',
          doc_count: 5,
          agg_1: {
            buckets: [
              {
                key: '0.0-5.0',
                from: 0,
                to: 5,
                doc_count: 0,
                agg_2: { count: 0, min: null, max: null, avg: null, sum: 0 }
              },
              {
                key: '5.0-10.0',
                from: 5,
                to: 10,
                doc_count: 5,
                agg_2: { count: 5, min: 7, max: 8, avg: 7.4, sum: 37 }
              }
            ]
          }
        }
      ]
    }
  }
}
```

Figure 37: Aggregation Results

It contains some metadata about how long it took the query to be executed, the number of hits it acquired etc. For our statistics rendering process we do not care about those data bits. What matters is the *aggregations* property that contains all the aggregation information. So, in order for the back-end to provide the statistics, it forwards the whole response again to the Wizard Controller, which will retain only the essential info.

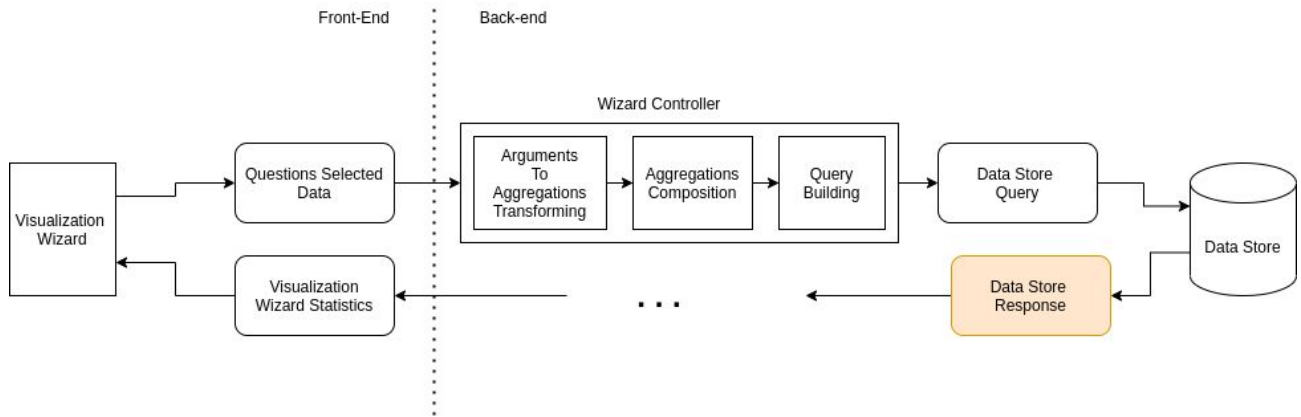


Diagram 23: Data Transformation Process - Phase 4

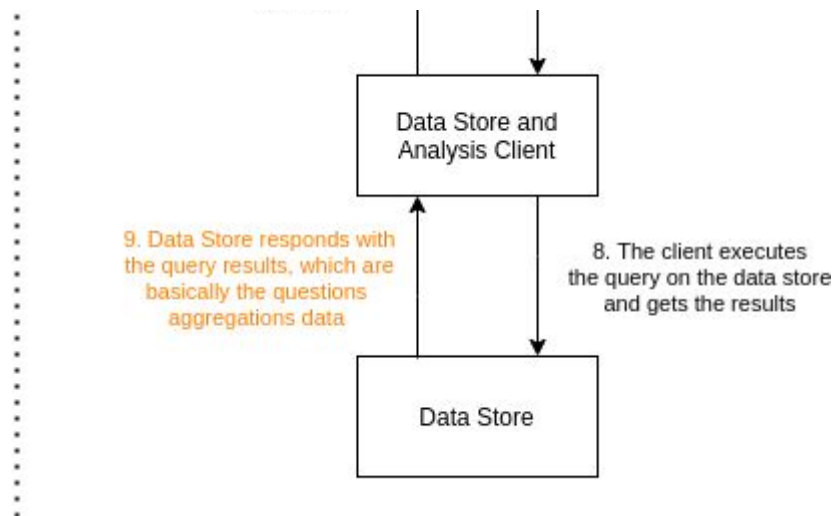


Diagram 24: Visualization Wizard Statistics Provider - Phase 4

E. Data Store Response Modifier

The control returns to the Wizard Controller with the Data-Stores response. In this phase, the Wizard Controller will keep only the basic information from the response and augment it with extra information, in order to make the rendering process the simplest possible.

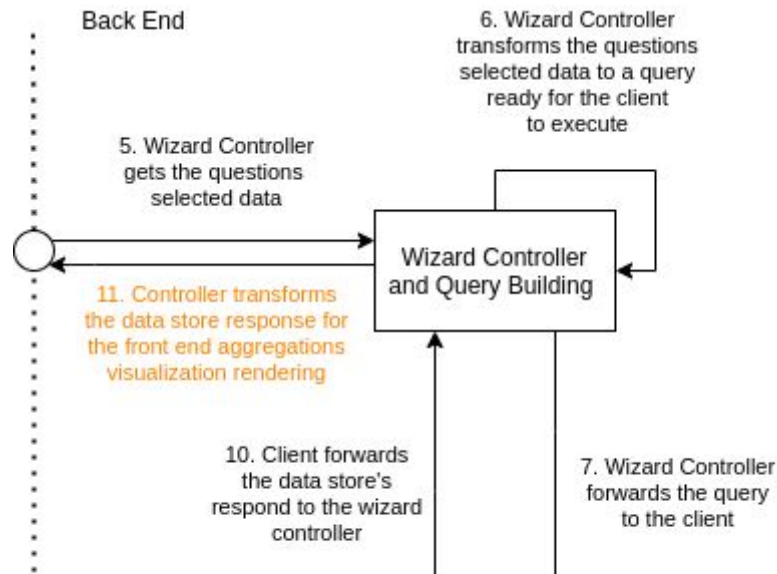


Diagram 25: Visualization Wizard Statistics Provider - Phase 5

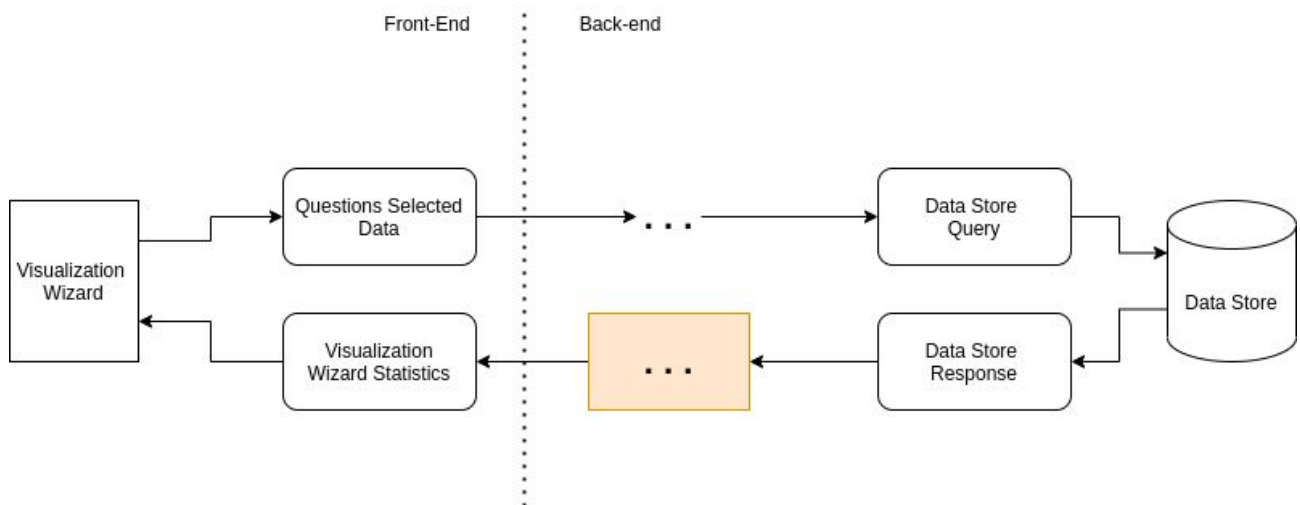


Diagram 26: Data Transformation Process - Phase 5

What the Front-End needs in order to render the results is the aggregations data and information about the question we aggregated our data to. The problem was that the data store response contained only the aggregations data without providing any details about the questions that aggregation was made for. So, for each aggregation we provide a set of information for the aggregation, like the question id, type etc. This pack of aggregations results, aggregations information and some other attributes are enough for front-end to visualize the wizard results. Let us preview how the wizard statistics look like, but focus on how these fields are created, and later in the last phase, we will explain what each one means.

```

wizardStats: {
  total_answers: 7,
  specializedAggrInfo: [ 'Answers on Question 2 should match Ναι \nΌχι \n' ],
  aggregations: {
    agg_0: {
      doc_count_error_upper_bound: 0,
      sum_other_doc_count: 0,
      buckets: [
        {
          key: 'Οικονομικά Κίνητρα',
          doc_count: 5,
          agg_1: {
            buckets: [
              {
                key: '0.0-5.0',
                from: 0,
                to: 5,
                doc_count: 0,
                agg_2: { count: 0, min: null, max: null, avg: null, sum: 0 }
              }, ...
            ]
          }
        }, ...
      ]
    }, ...
  ],
  agg_info: [
    {
      question: {
        q_id: 'Q01',
        q_type: 'SetOfStrings',
        q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
      },
      forTheQuestion: 'Question 1',
      isFinal: false
    }, ...
  ]
}

```

Figure 38: Wizard Statistics

The *total answers* field is the number of answered surveys we have for the specific survey, the *specializedAggrInfo* concerns non-general terms questions and indicates the answer fields that the results should or must match. The *aggregations* contain all the information about the aggregations that happened to the questions. It recursively contains the set of aggregations that were applied in the sequence of the questions. The last one, the *agg_info* is actually the Questions Selected Data. Because the information is already there, we just create a way for each aggregation to correspond to the right aggregation data. That happens by the aggregation name (e.g. “agg_0” and the question's data index in the Questions Selected Data array).

The code that modifies the data store's response and creates the above result:

```

/*
    For each question parameter from the front-end,
    augment the elastic respond to help the front render the data
    by telling which aggregation was which question
*/

let agg_info = [];
let specializedAggrInfo = [];
q_params.forEach((question, index) => {
    let isFinal = index + 1 === q_params.length;
    switch (question.q_type) {
        case "Number":
            agg_info.push({
                question: question,
                forTheQuestion: `Question ${index + 1}`,
                isFinal: isFinal,
            });
            break;
        default:
            if (question.q_args.isAggrGeneral) {
                agg_info.push({
                    question: question,
                    forTheQuestion: `Question ${index + 1}`,
                    isFinal: isFinal,
                });
            } else {
                let specializedParamsOnQuestion = "";
                switch (question.q_args.aggregateOp) {
                    case "must":
                        specializedParamsOnQuestion += `Answers on Question ${
                            index + 1
                        } must match `;
                        question.q_args.aggregateOn.forEach((value) => {
                            specializedParamsOnQuestion += `${value} \n`;
                        });
                        break;
                    default:
                        specializedParamsOnQuestion += `Answers on Question ${
                            index + 1
                        } should match `;
                        question.q_args.aggregateOn.forEach((value) => {
                            specializedParamsOnQuestion += `${value} \n`;
                        });
                }
                specializedAggrInfo.push(specializedParamsOnQuestion);
            }
    }
});
response_data = {
    total_answers: result.hits.total.value,
    specializedAggrInfo: specializedAggrInfo,
    aggregations: result.aggregations,
    agg_info: agg_info,
};

```

Code 19: Wizard Statistics Building

The above code follows an iterative logic of getting each question and building the aggregation info with it.

Finally, we can proceed to the next phase, in which we will close the circle the Wizard Statistics providing .

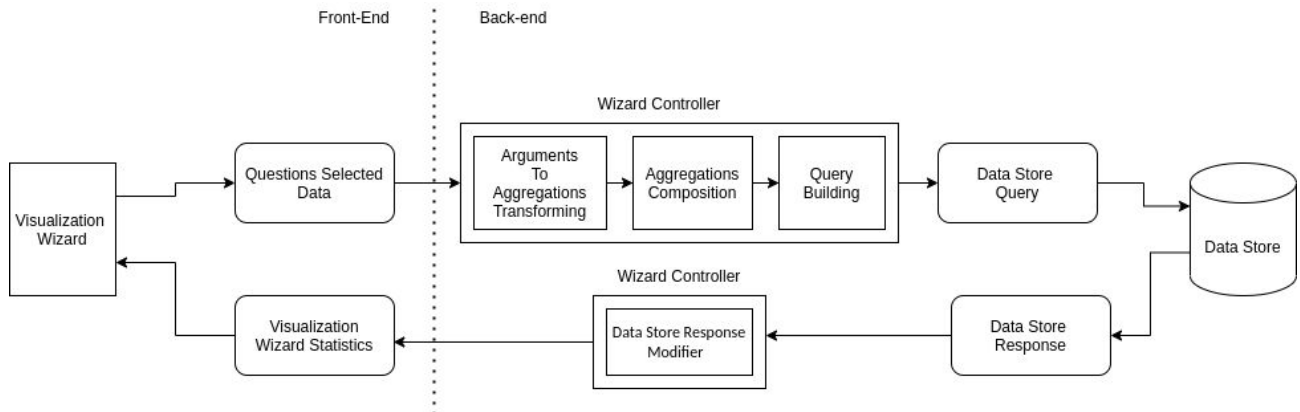


Diagram 27: Data Transformation Process - Backward Complete

F. Wizard Statistics

The last part of the Statistics Provider process is to demonstrate the Wizard Statistics once again and see how each value is will be used later on the Visualization rendering process. From this point, the Visualization Wizard page can retrieve the Wizard Statistics.

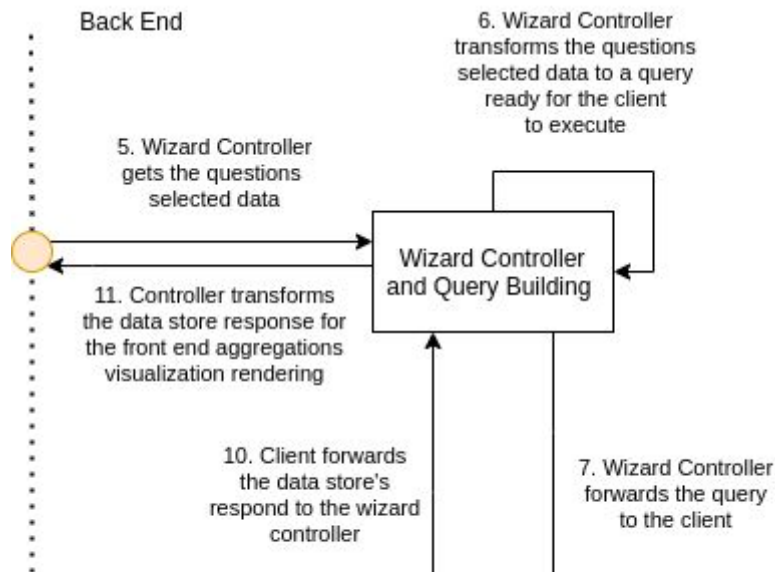


Diagram 28 : Visualization Wizard Statistics Provider - Phase 6

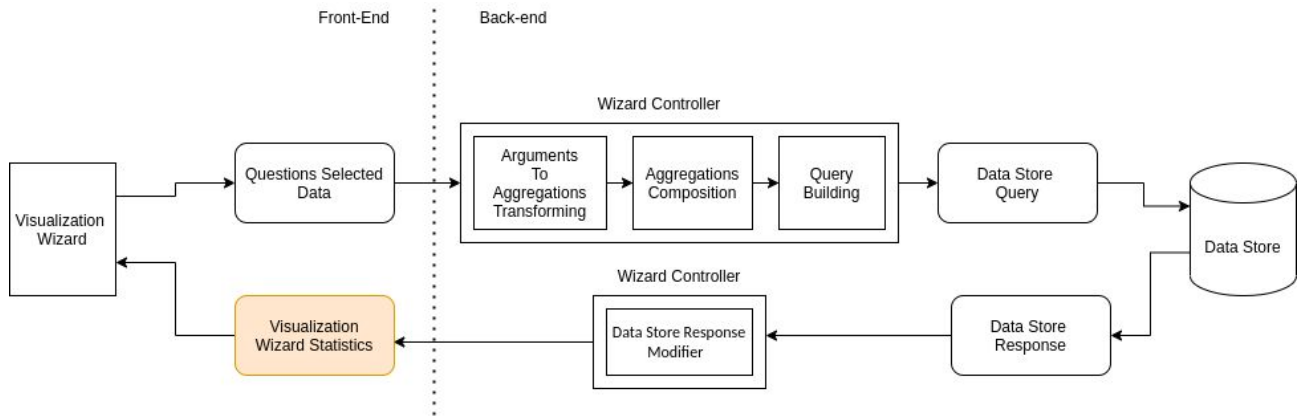


Diagram 29: Data Transformation Process - Phase 6

As we have already shown , the wizard statistics look like this:

```
wizardStats: {
  total_answers: 7,
  specializedAggrInfo: [ 'Answers on Question 2 should match Naf \n0x1 \n' ],
  aggregations: {
    agg_0: {
      doc_count_error_upper_bound: 0,
      sum_other_doc_count: 0,
      buckets: [
        {
          key: 'Οικονομικά Κίνητρα',
          doc_count: 5,
          agg_1: {
            buckets: [
              {
                key: '0.0-5.0',
                from: 0,
                to: 5,
                doc_count: 0,
                agg_2: { count: 0, min: null, max: null, avg: null, sum: 0 }
              }, ...
            ]
          }, ...
        ]
      }, ...
    ],
    agg_info: [
      {
        question: {
          q_id: 'Q01',
          q_type: 'SetOfStrings',
          q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
        },
        forTheQuestion: 'Question 1',
        isFinal: false
      }, ...
    ]
  }
}
```

The front-end will acquire each aggregation and the info which it accompanies, in order to visualize each aggregation bucket. For the non-general terms aggregations we provide from the back-end the answers that the aggregations happened for, like which answers should match or must match, and provides the total survey answers for the statistics to be complete.

Having clarified the Statistics Provider, the whole process is summarized with the following diagram:

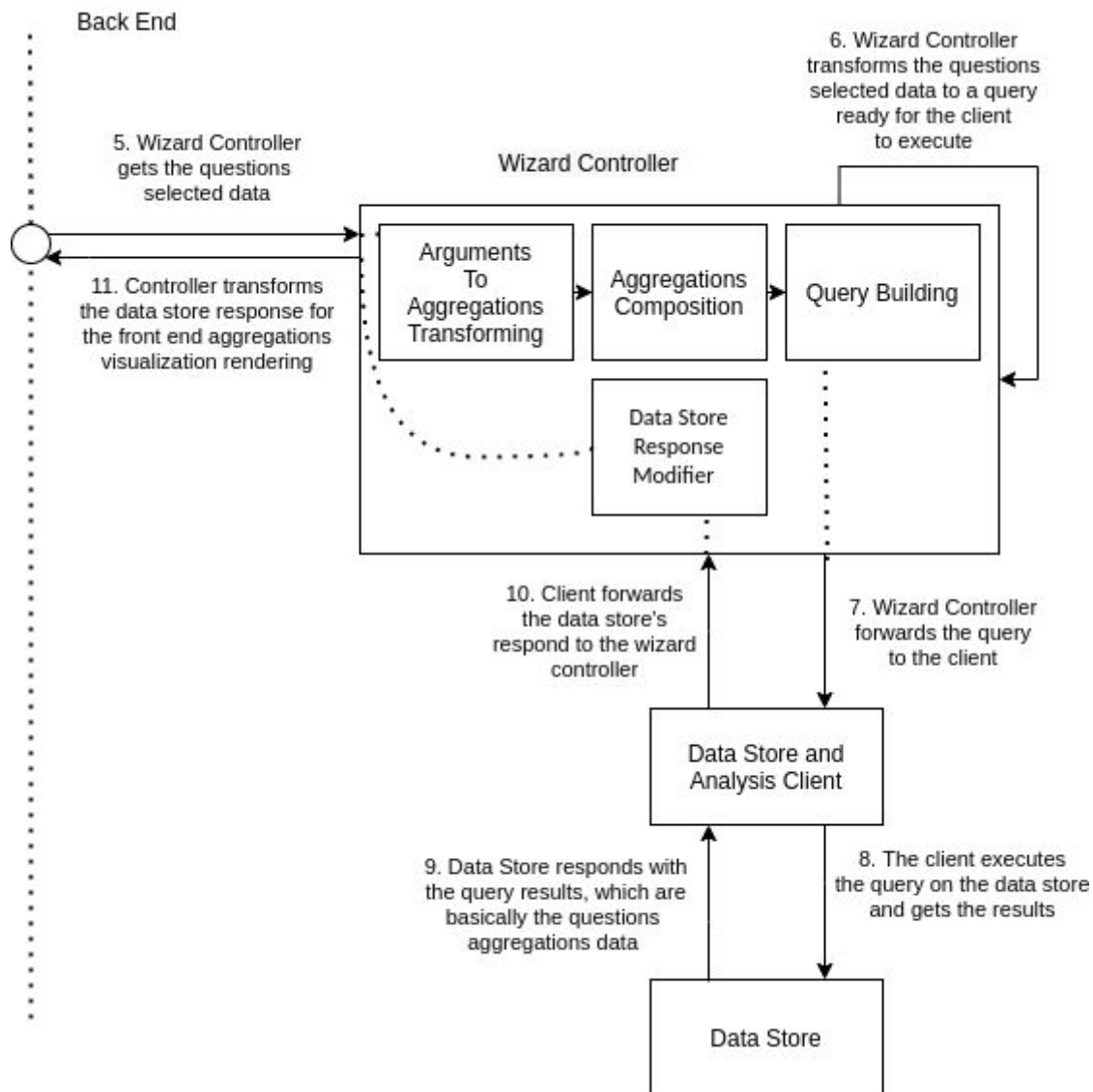


Diagram 30: Visualization Wizard Statistics Providing Complete

Now we can proceed to the Visualization Wizard Rendering process and clarify this as well.

II. Visualization Wizard Renderer

Okay, the whole Statistics Providing has occurred . There are **2** core issues concerning the whole Visualization Wizard process. The first is how the front-end constructs the Questions Selected Data and the second how it renders the Wizard Statistics. Before explaining each issue, let us see how the whole rendering process in front-end is:

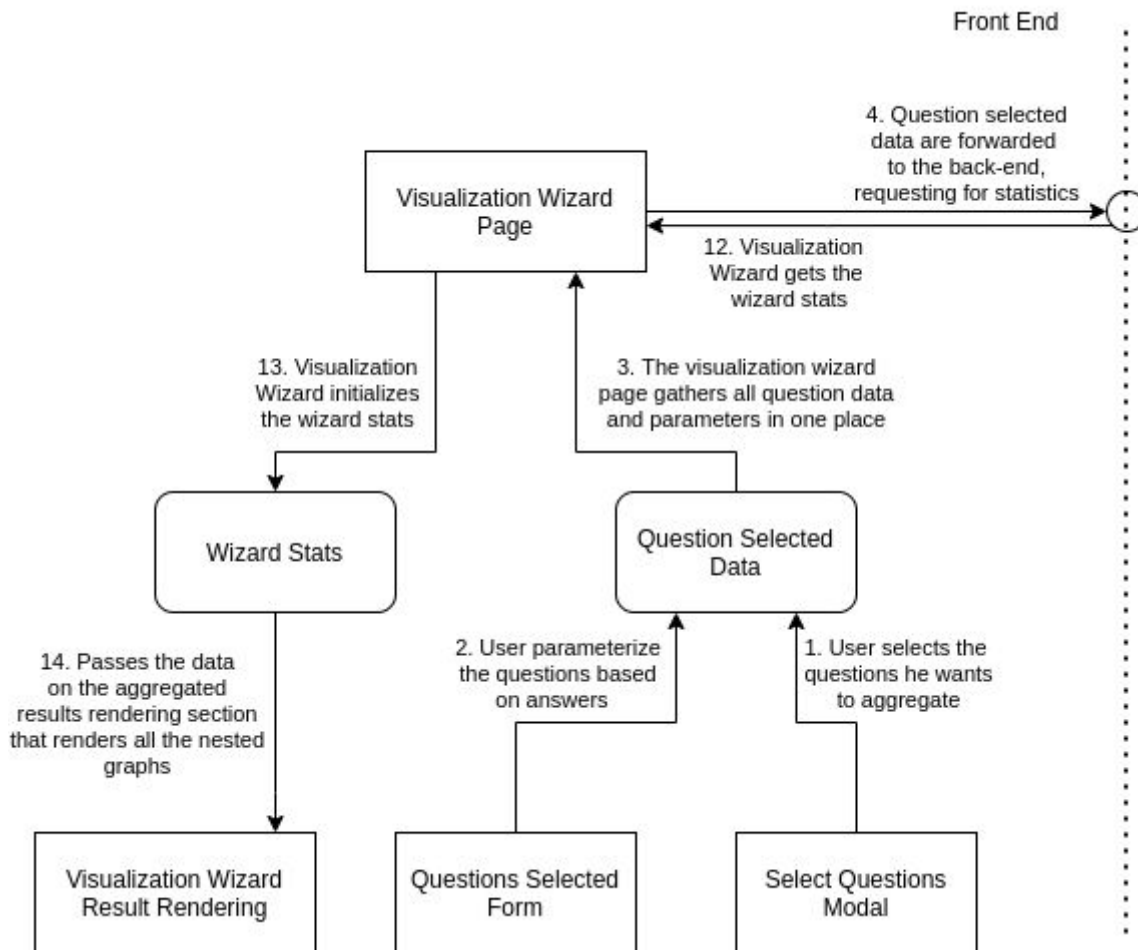


Diagram 31: Visualization Wizard Renderer Abstract

The front-end process starts by the user selecting the questions which they want to set to the wizard, then parameterizes them and lastly renders the results after they have been provided to the back-end. In this chapter we will focus on the two trivial issues. The Questions Selected Data building from the front-end and the Wizard Stats Visualization. How the user selects the questions is up to React logic which will be explained in the technical analysis chapter. What is important in these phases is to understand that the user selects questions by a modal UI component.

i) Questions Selected Data Building

We will start off by showing another screen from the Visualization Wizard before asking the Wizard to get the statistics.

DEV STUDY

Home / Pick Results / Ερωτηματολόγιο Φοιτητών

Visualization Wizard

Play with the survey results

Based on: Question 1

Nai : 4	▼
Όχι : 3	▼

Total answers: 7

For the questions:

Question 1

Σκοπεύετε να μεταβείτε στο εξωτερικό για περαιτέρω σπουδές;

❤️ 📊 ▼

Question 2

Βαθμολογήστε την ποιότητα των συνηθειών σας (διατροφή, άσκηση, ποιότητα ύπνου...).

❤️ 📊 ▼

Calculate:

Question 3

Οι σπουδές σας πόσο τον άλλαξαν;

❤️ 📊 ▼

Dev Study Goals ?

Trying to understand programmers' practices, principles and notions in the workspace, and how people from different backgrounds transit and adapt on it.

You may also see :

- About conductors
- More about our surveys
- Terms and Conditions

© 2020 Copyrighted Survey

Figure 39: Questions Selected Data Building

The questions selected data for the above example should look like this:

```
[
  {
    q_id: 'Q26',
    q_type: 'String',
    q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
  },
  { q_id: 'Q29', q_type: 'Number', q_args: { range: [ 0, 5, 10 ] } },
  {
    q_id: 'Q32',
    q_type: 'String',
    q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
  }
]
```

The above data format that the front-collects happens in different levels. The whole diagram that outlines the process is the following:

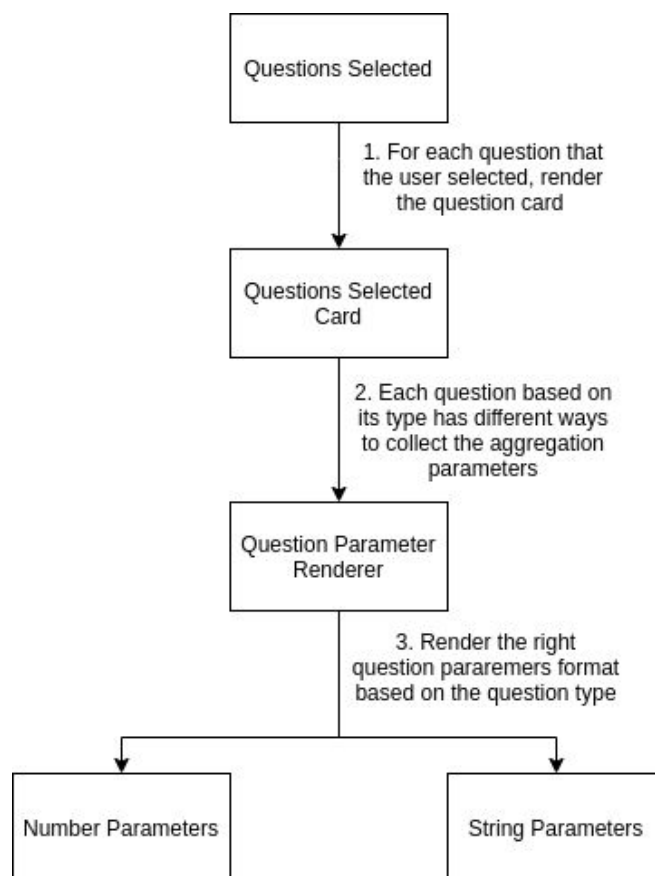
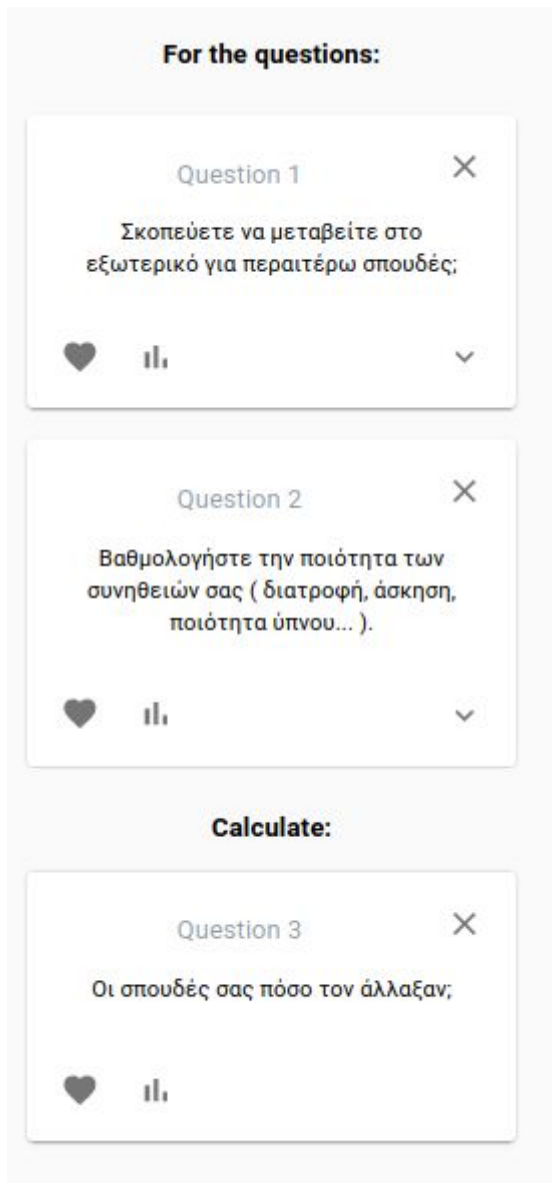


Diagram 32: Visualization Wizard Renderer Component Tree

Each rectangle represents a component for the front-end technologies. Let us see them one by one with an example, code and design details.

Questions Selected

Questions Selected Component contains the following part of the UI:



Each Question here is represented by a Question Selected Card UI component, which we will next see. It is important to mention here that only the last question requires different handling, because this is the question we plot the data for.

The Questions Selected component only needs the questions that the user has selected.

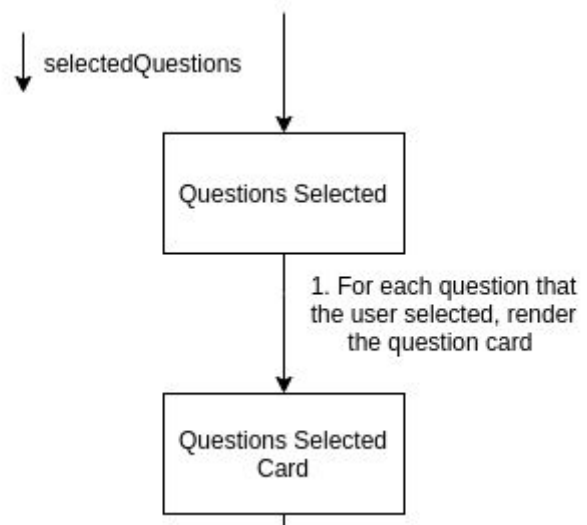


Figure 40: Questions Selected Rendering Cards

The Questions Selected component gets as a property the questions which the user has selected and renders for each one a Question Selected UI Card. Let us see the Question Selected Card details.

Question Selected Card

Let us isolate two cards and analyze them.

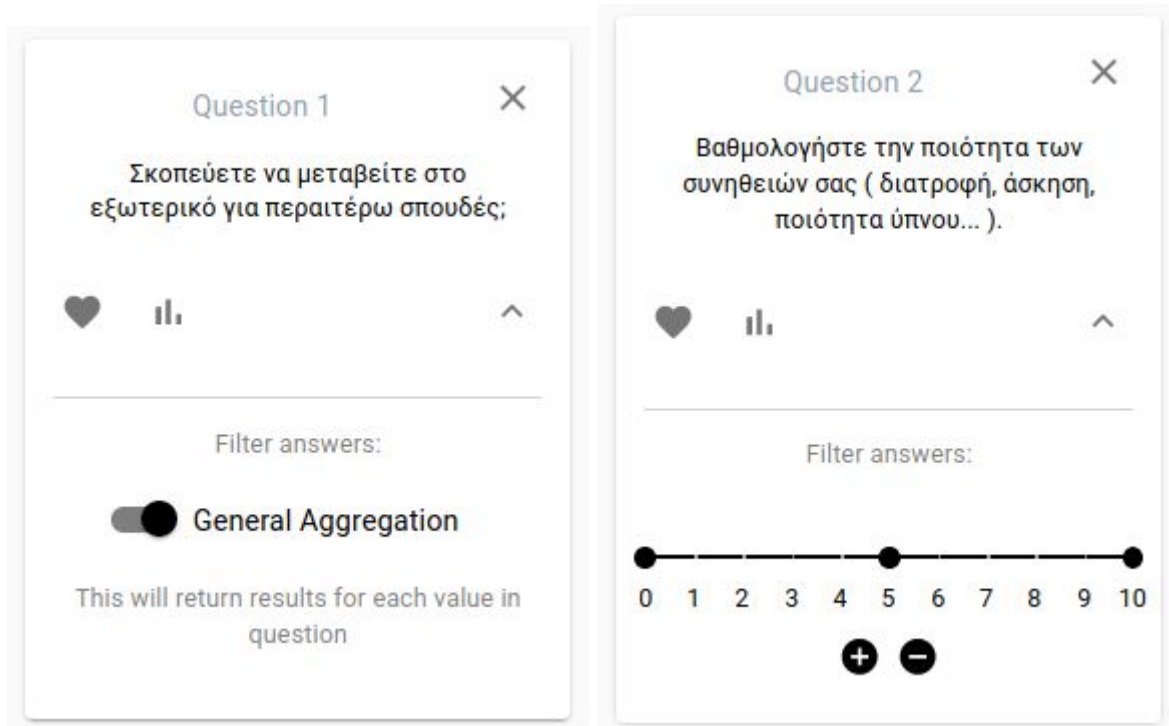


Figure 41: Questions Selected Cards

Each card is divided into two parts. The Questions data part, which contains the Questions text and Questions Parameters parts that we will soon see. The Question Selected Card needs as an argument the question data, in order to be able to render it.

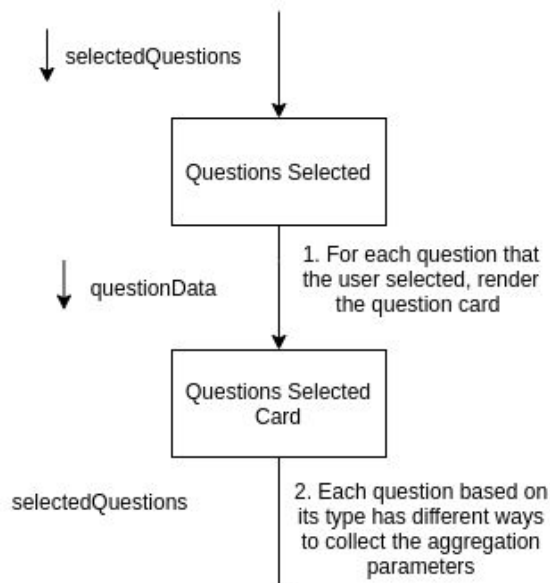


Diagram 33: Passing Question Data

The Question Selected Card needs the right type of parameters. We have seen how the Question's Selected Data looks like. The question's aggregation parameters depend on the question type. The above question card has different question parameters, as you can see, and a different form of input. Let us focus on this part:

Filter answers:

☐ General Aggregation

Aggregate On:

☒ Nai

☒ Oxi

Answers:

☐ Should Match

☒ Must Match

0 1 2 3 4 5 6 7 8 9 10

+ -

Figure 42: Questions Card Parameters

The Question Selected Card in order to render the right question parameters form uses the Question Parameter Renderer component, by passing it the `questionData` which it already has.

Question Parameter Renderer

The question parameter renderer defines which UI question parameter form to render by conditionally rendering the right form based on the question type.

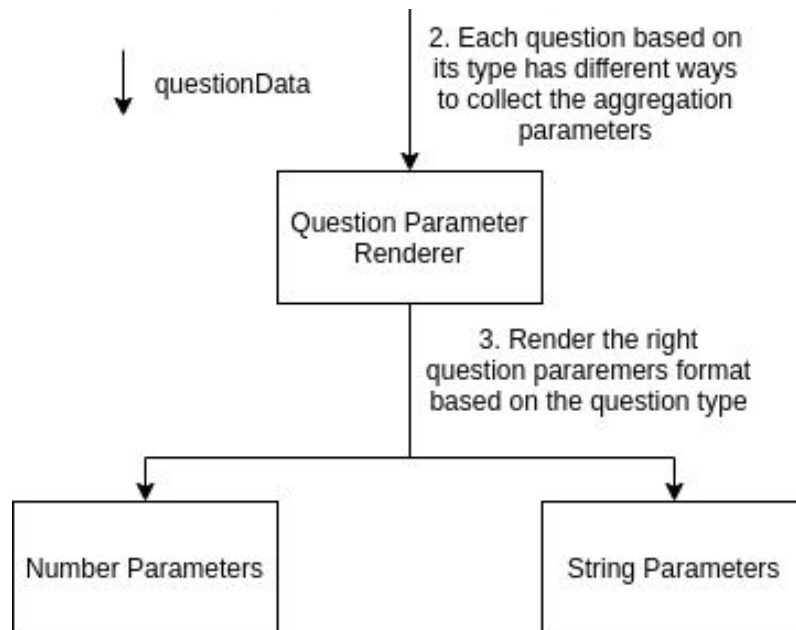


Diagram 34: Rendering Aggregation Parameters

The code of the Question Parameter Renderer is the following:

```
import React from "react";
import {
  NumberParams,
  StringsParams,
} from "../ParametersFormComponents/ParametersFormComponents";

const paramsForm = (props) => {
  const questionData = props.questionData;
  if (!questionData) return null;
  var inputElement;
  /* Based on the question type create the proper visualization */
  switch (questionData.type) {
    case "Number":
      inputElement = (
        <NumberParams questionData={questionData} index={props.index} />
      );
      break;

    default:
      inputElement = (
        <StringsParams questionData={questionData} index={props.index} />
      );
  }
  return (
    <div>
      <p style={{ color: "gray" }}>Filter answers:</p>
      {inputElement}
    </div>
  );
};

export default paramsForm;
```

Code 20: Form Aggregation Parameters Renderer

Based on the question type, the component decides to render the proper question aggregation parameter form. The same conditional logic is clearly seen through the whole code, making the implementation logic consistent across the app. For the final part let's see the different Question Form Parameters that the Question Parameter Renderer produces.

Question Parameter Forms

We have now reached the last components group which is the Question Parameters Form components. So far, we have 2 types of form components. Let us see them both:

- **Terms Questions:** The term questions need the following question parameters:

Filter answers:

☐ General Aggregation

Aggregate On:

☒ Nai

☒ Oxi

Answers:

☐ Should Match

☒ Must Match

The question parameters are stored in the String Params component of our code which gathers the information to the following data structure format:

```
{
  aggregateOn: ["Nai", "Oxi"],
  aggregateOp: "must",
  isAggrGeneral: false,
};
```

Figure 43: Terms Questions Form Parameters

The code which implements the above behavior is just a React component that uses checkboxes, radio buttons and event handlers to keep the data up to date. The same applies for the following question form parameters type.

- **Stats Questions:** The stats questions need the following question parameters:

And the data which this form produces are stored on this object format:

Filter answers:

0 1 2 3 4 5 6 7 8 9 10

☒ ☐

```
{
  range: [0, this.ceil / 2, this.ceil],
};
```

Figure 44: Stats Questions Form Parameters

Now that all the questions selected have been rendered to the end and we have a way to update the data for each, let us see how all these are gathered to a central place.

Let us see how the whole process of the Questions Selected Data rendering looks like:

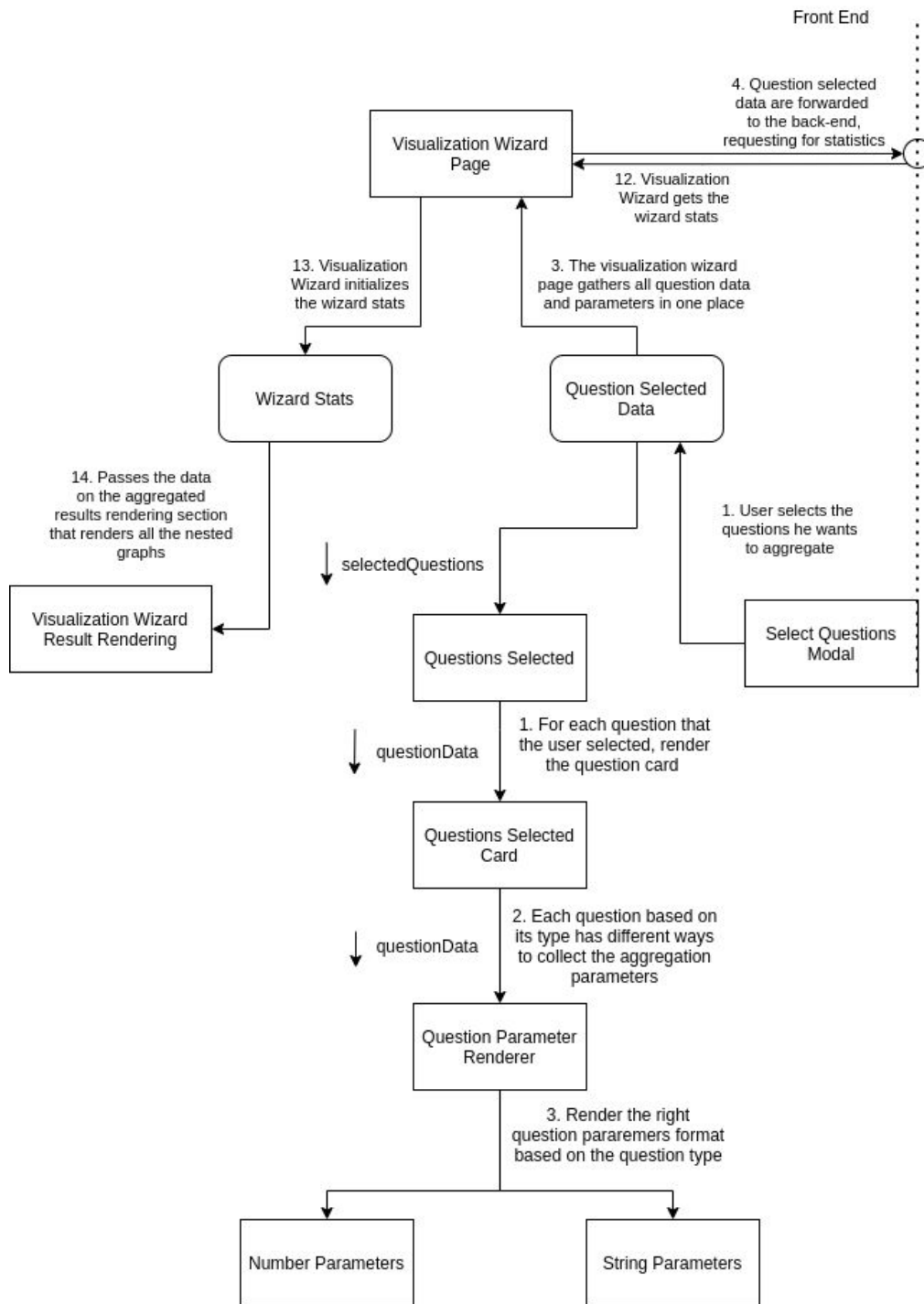


Diagram 35: Visualization Wizard Front-End Detailed

All of the above rendering processes create the Questions Selected Data step by step, by taking the basic information like the questions selected id,type and adding to them the aggregation parameters that the user defines. The Number Parameters and String Parameters update the Questions Selected Data arguments each time a value changes, despite the fact of these deep component-like structures. (More on Technical Analysis).

ii) Wizard Stats Visualization

The last part of the whole visualization process is the rendering of the wizard statistics. We have seen exhaustively how the front-end gathers the Questions Selected Data and how it retrieves the wizard statistics. It is now time to see how the visualization of the Wizard Statics works. This procedure happens in the Visualization Wizard Result Rendering component. Before jumping into details, let us remember how the wizard statistics looks like:

```
wizardStats: {
  total_answers: 7,
  specializedAggrInfo: [ 'Answers on Question 2 should match Ναι \nΌχι \n' ],
  aggregations: {
    agg_0: {
      doc_count_error_upper_bound: 0,
      sum_other_doc_count: 0,
      buckets: [
        {
          key: 'Οικονομικά Κίνητρα',
          doc_count: 5,
          agg_1: {
            buckets: [
              {
                key: '0.0-5.0',
                from: 0,
                to: 5,
                doc_count: 0,
                agg_2: { count: 0, min: null, max: null, avg: null, sum: 0 }
              }, ...
            ]
          }
        }, ...
      ]
    }, ...
  ],
  agg_info: [
    {
      question: {
        q_id: 'Q01',
        q_type: 'SetOfStrings',
        q_args: { aggregateOn: [], aggregateOp: 'must', isAggrGeneral: true }
      },
      forTheQuestion: 'Question 1',
      isFinal: false
    }, ...
  ]
}
```

The Wizards Result rendering process will render the total number of answered surveys, the special aggregation info that we have for the non-general questions aggregations, like

what question's values the answers should or must match and the question aggregation buckets along with their statistics and information. The end result given can look, for example, like this:

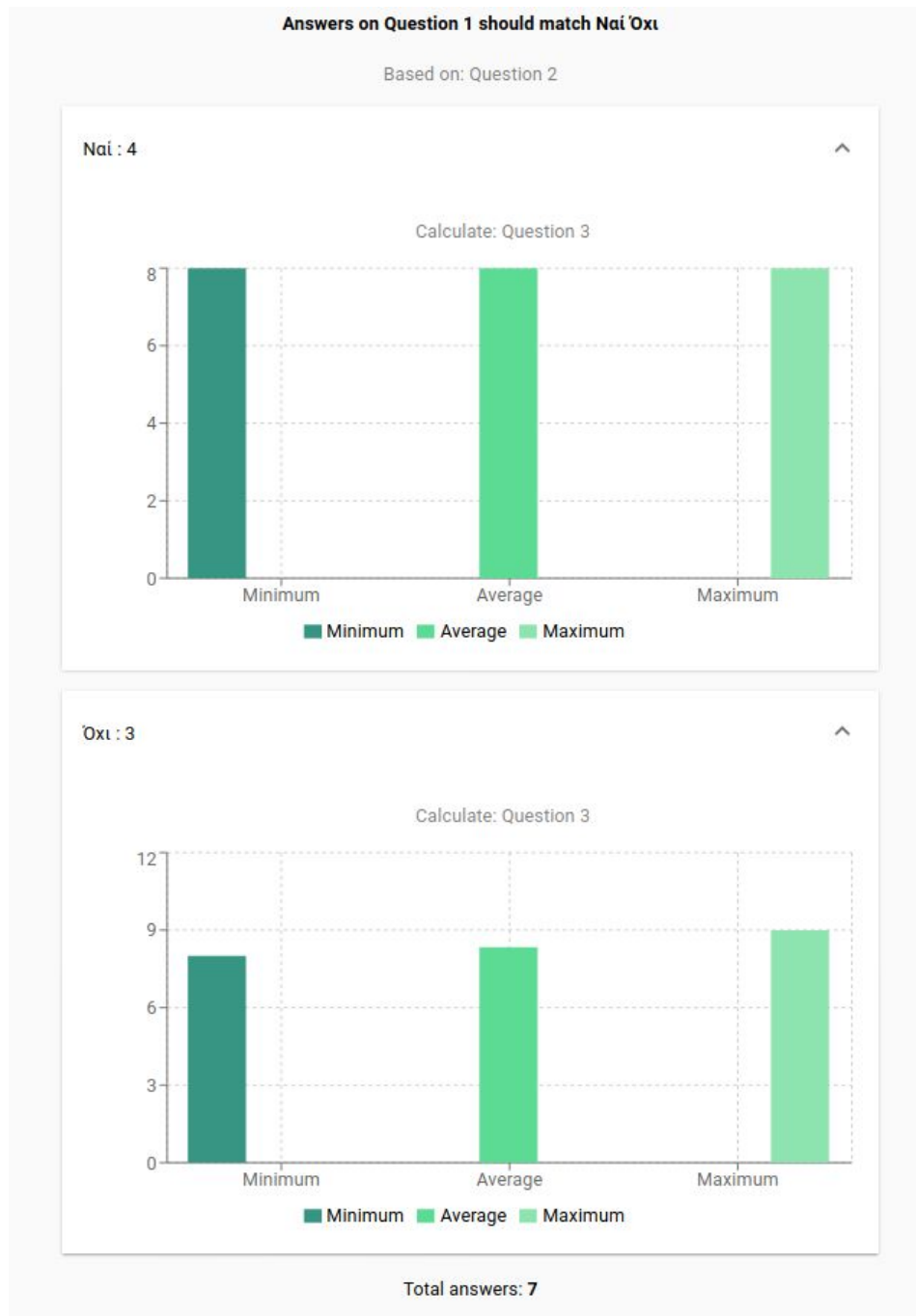


Figure 45: Visualization Wizard Another Example

The code which accomplishes all these nested-like visualization processes is the following:


```

import React from "react";
import Visualization from "../../VisualizationRenderer/VisualizationRenderer";
import "@material-ui/core";
import "../WizardResults.css";

// This is a recursive method that will produce
// the visualizations, based on backends response
const visualizeStats = (aggr, aggrNum, aggr_info) => {
  let aggData = [];
  aggData.push(
    <p>Based on: {aggr_info[aggrNum].forTheQuestion}</p>
  );
  // If bucket contains another aggregation, move towards that aggr
  if (!aggr_info[aggrNum].isFinal) {
    for (var i = 0; i < aggr.buckets.length; i++) {
      aggData.push(
        <Accordion>
          <AccordionSummary
            expandIcon={<ExpandMore />}
            aria-controls="panella-content"
            id="panella-header"
          >
            {aggr.buckets[i].key} : {aggr.buckets[i].doc_count}
          </AccordionSummary>
          <AccordionDetails className="alignCenter">
            {visualizeStats(
              aggr.buckets[i]['agg_${aggrNum + 1}'],
              aggrNum + 1,
              aggr_info
            )}
          </AccordionDetails>
        </Accordion>
      );
    }
  }
  // If bucket doesn't have a new aggregation, it means we are on
  // the last answer, so, do calculate the graph
  else {
    let total = 0;
    var stats;
    switch (aggr_info[aggrNum].question.q_type) {
      case "Number":
        stats = aggr;
        break;
      default:
        stats = aggr.buckets;
    }
    return (
      <div>
        <p> Calculate: {aggr_info[aggrNum].forTheQuestion}</p>
        <Visualization
          questionData={{ type: aggr_info[aggrNum].question.q_type }}
          questionStats={{ total: total, stats: stats }}
        />
      </div>
    );
  }
  return aggData;
};

export default function WizardResults(props) {
  var result = visualizeStats(props.stats.aggregations.agg_0, 0, props.stats.agg_info);
  return (
    <div>
      {props.stats.specializedAggrInfo.map((extraParams) => {
        <h4>{extraParams}</h4>
      })}
      {result}
      <p> Total answers: <b>{props.stats.total_answers}</b></p>
    </div>
  );
}

```

Code 21: Visualization Wizard Stats Visualizer

The code in order to render the nested aggregations implements a recursive function that follows the same logic that the aggregations do. For each aggregation bucket it creates an Accordion UI component. It gets the data from the aggregation info stats structures, so it outputs all the information for each bucket. Lastly, for the last question, that all the aggregations happened for it passes the statistics to the Visualization Renderer that we saw on 2.4.1.1 chapter. Then the final graph is rendered for that question. This is how the Visualization Wizard renders the results.

Let us finalize the Visualization Wizard Renderer with the last diagram and then sum up the whole visualization wizard process.

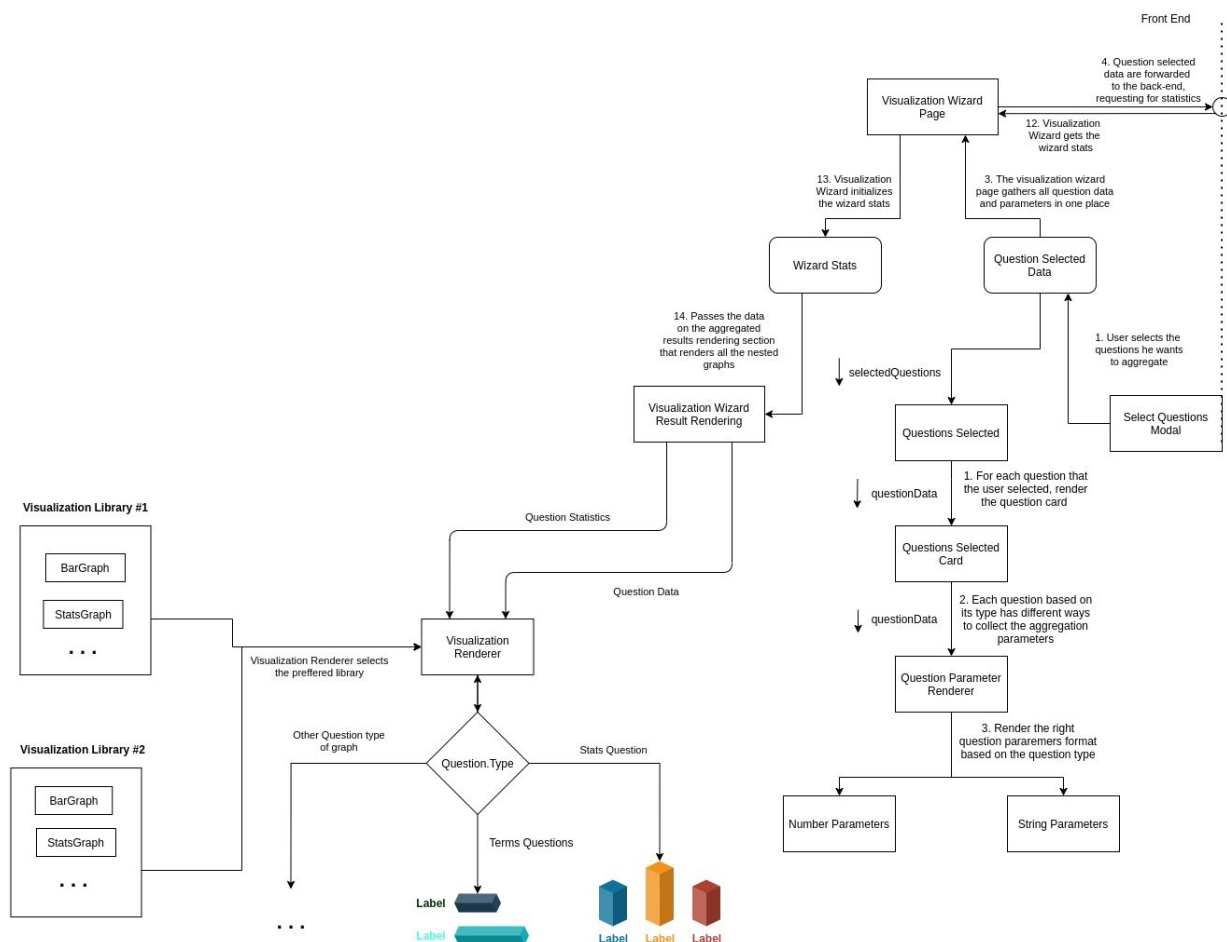


Diagram 36: Visualization Wizard Front End Complete

Each of the above processes has been described in the chapters above.

III. Summing up

To sum up the whole visualization wizard process, I will present the last diagram, which includes all the different steps which were documented throughout this thesis chapter.

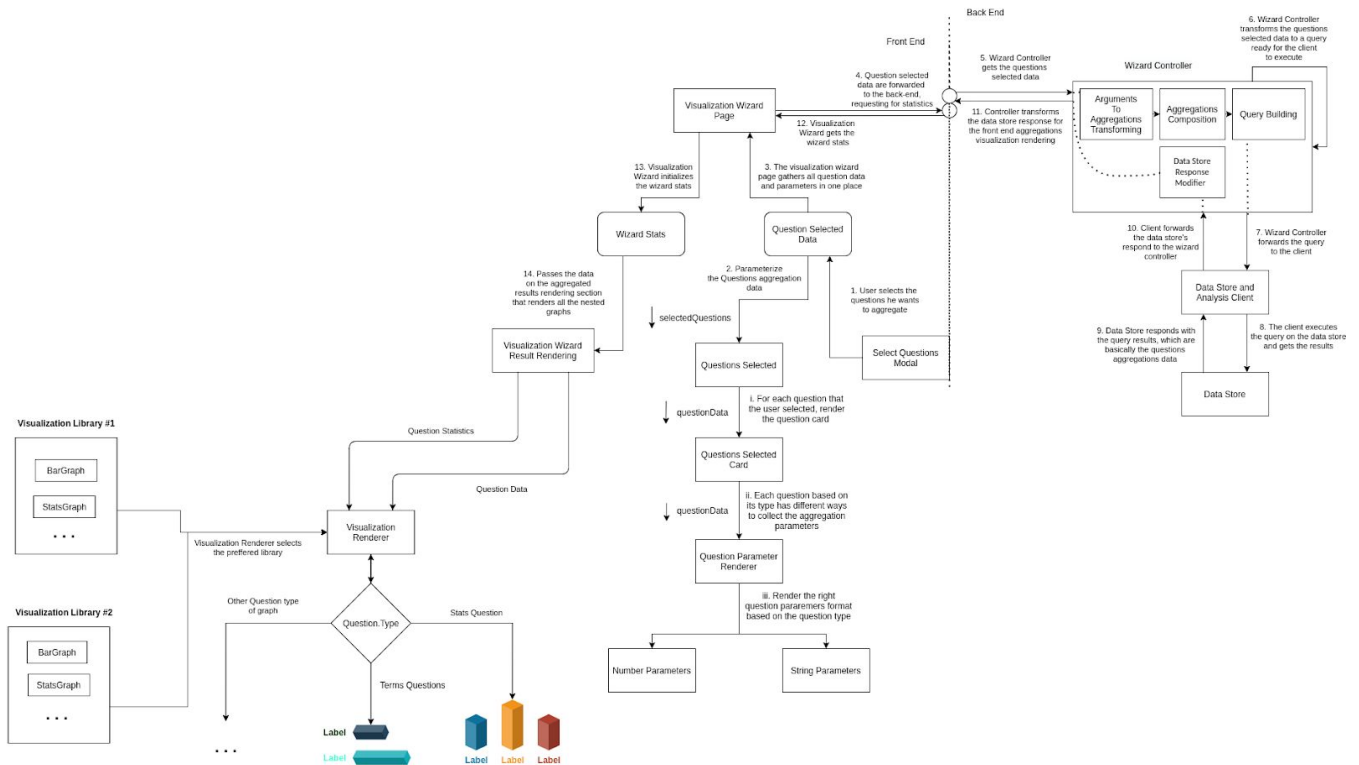


Diagram 37: Visualization Wizard Complete

The Visualization Wizard has now been completed. And it was kind of huge. It required the complete understanding of the technologies used, enacted numerous transformations from one format to another and applied a different set of programming techniques. If the initial goal was not clear enough, and without having fully comprehended the different processes, the Visualization Wizard would not be able to be completed. But in the end it was worth it, because now users will be able to explore the results and try to 'mine' a set of information from the surveys.

The next steps are to add more functionality, like download results, or add more aggregation types, new diagrams etc. Also, refactor the different code parts, optimize the performance and furthermore .

With this , the core features that our SaaS offers has now been completed. It is time to move on to more lightweight items and then proceed to the technical analysis.

3.5 UX and UI decisions

In this chapter we will take a look at the User Experience and Interface design decisions and how the general platforms aesthetic was created.

3.5.1 Usability Evaluation

Designing a simple, usable and minimal user interface and a pleasing user experience is of top priority in any computer system. Our platform follows an intuitive design that provides ease of learning and efficiency of use. Each design approach follows the 10 usability rules of Nielsen, like the visibility of system status, indicating each time when a service is loading. It matches the real world, when it comes to intuition, as each survey contains questions that we try to represent in the real world, like order etc. The general aesthetics of the platform follows specific palettes and design behavior, in order for it to be consistent across the different pages.

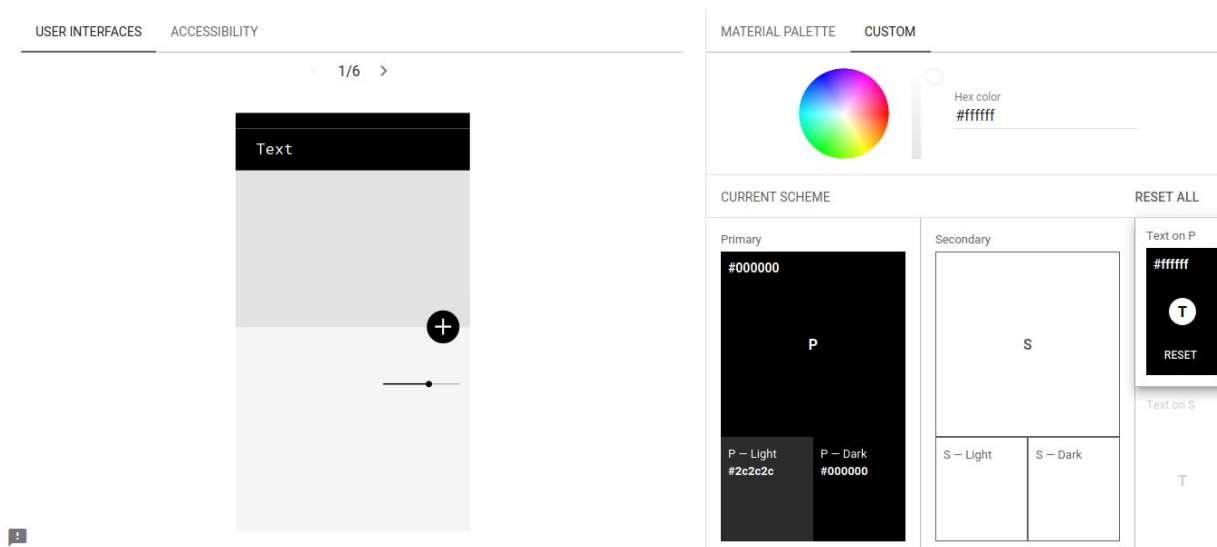


Figure 46: Theming and Color Palettes

It also offers Error Prevention and Recovery,

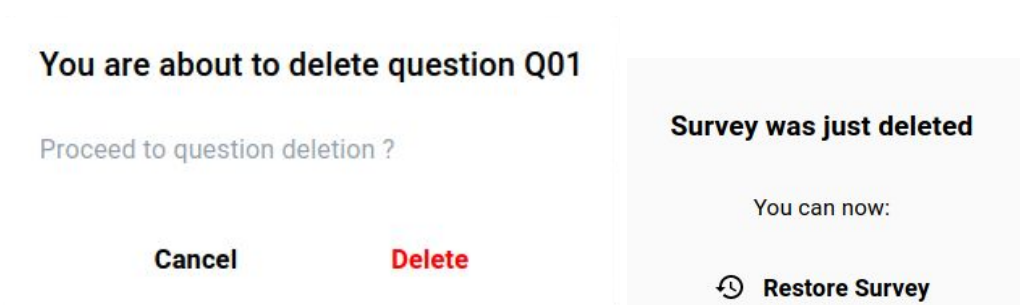


Figure 47: Error Prevention Options

While also applying a different set of methods and rules, like the time to acquire a target is a function of the distance to and size of the target (Fitts Law) or the time it takes to make a decision increases with the number and complexity of choices (Hick's Law) and another set of fundamental UX rules that aim for the maximum platform learnability and user satisfaction interacting with our product.

The platform also uses eye-candy icons and graphical resources reinforcing the user's experience and making the platform more appealing. Graphical assets used:

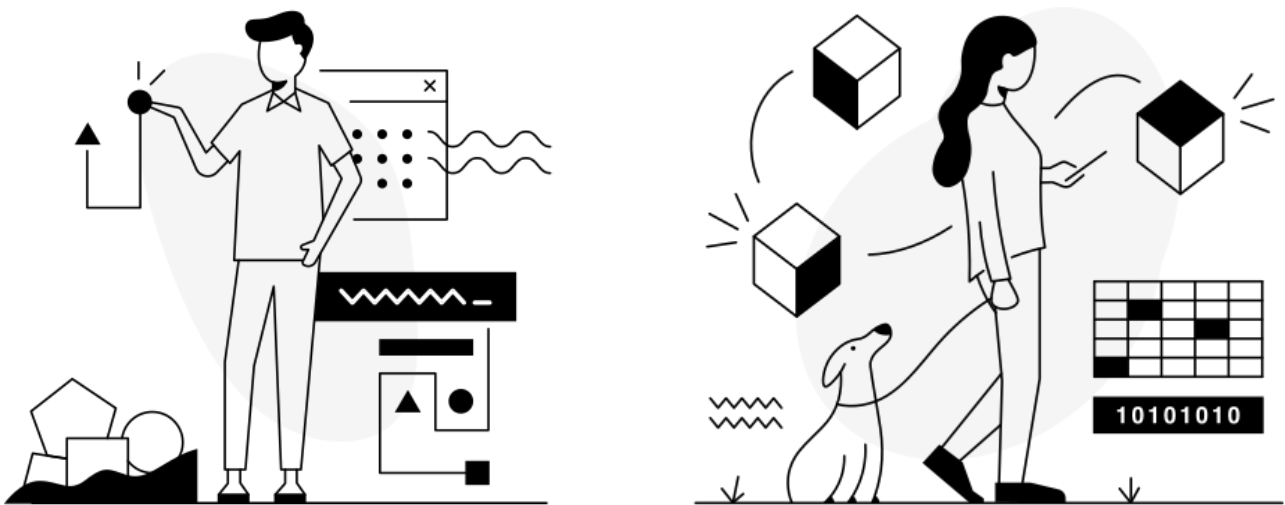


Figure 48: Graphic Designs

And icons provided from [Flaticon](#) and [Material Icons](#).

Finally, the whole UX and UI character is based upon [Material Design](#). The majority of web platforms use Material design, as it is widely accepted and fits the general platform intentions and initial purpose.

3.5.2 Theming

The platform provides 2 themes for the use to choose from. The Light theme and the Night theme. Both themes were created with the help of the Material library that we will explain in the Technical Analysis. The user can switch between the two themes in the navbar.

The two theme outcomes are the following:

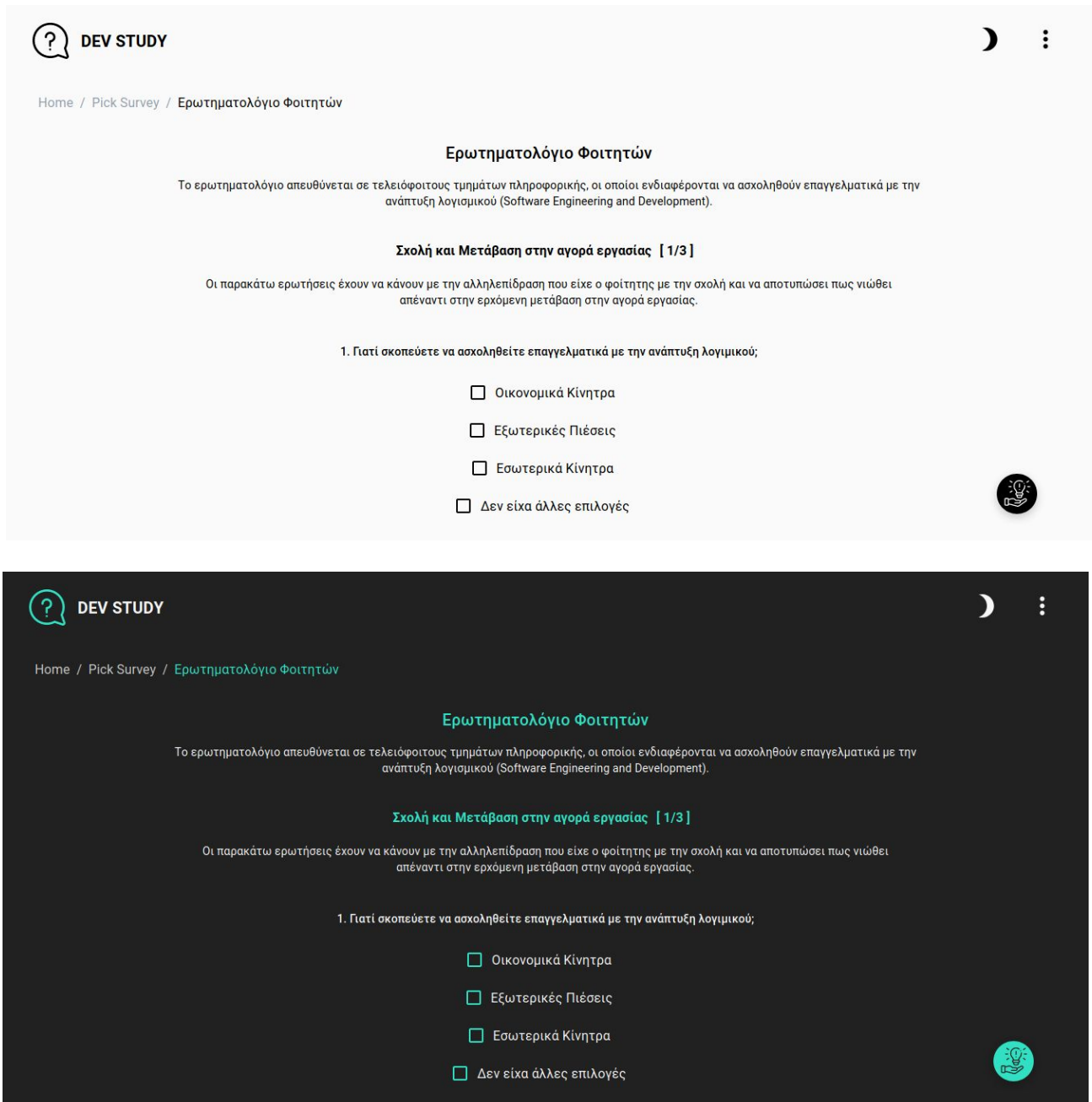


Figure 49: Light And Dark Theme

It is also important to mention that it is easy for the devs to create a wide range of themes for the users to choose from.

3.5.3 Accessibility

The tools which are used throughout our application ensure that the surveys and the results will be accessible for everyone. MaterialUI provides [Accessibility Features](#) that follow the WCAG guidelines, the app is passed through accessibility evaluation phases and the color scheme complies with the accessibility guidelines. It will be one of the top priorities to apply the Design for All principles, so that anyone can have access and be a survey participant or explore the survey results.

3.6 Design Challenges

Designing was the most important part for the whole implementation to be completed. We needed to design how the front-end will look like and the whole user interface layout and behavior, how the back-end will be architected in order to serve requests, how the extra tools and frameworks will comply with the whole base architecture, how the data will be stored in our systems and many others. All that for the distinct entities to communicate harmoniously and be as stable as possible. Each one of the above designing phases came along with a set of challenges that were faced and each one required clean and concrete design.

3.6.1 Architectural Decisions

The architectural decisions that were suggested were made after a series of design challenges. The most important of them were:

- *What would be the layout of each page:*

Each page layout and structure was defined after creating wireframes, from low to high fidelity, so that it would be clear for the later coding phase.

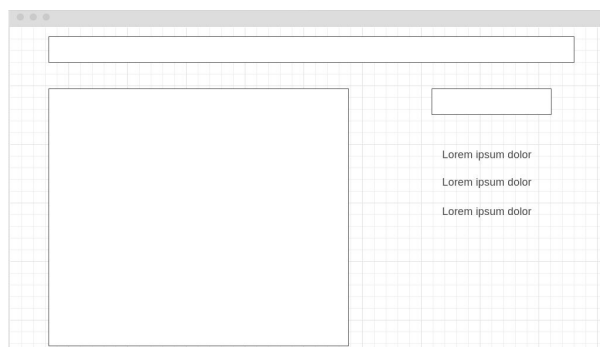


Figure 50: Wireframes

- *What is the front-end structure ?*

The front-end structure is component-based and follows a hierarchical tree-like logic.

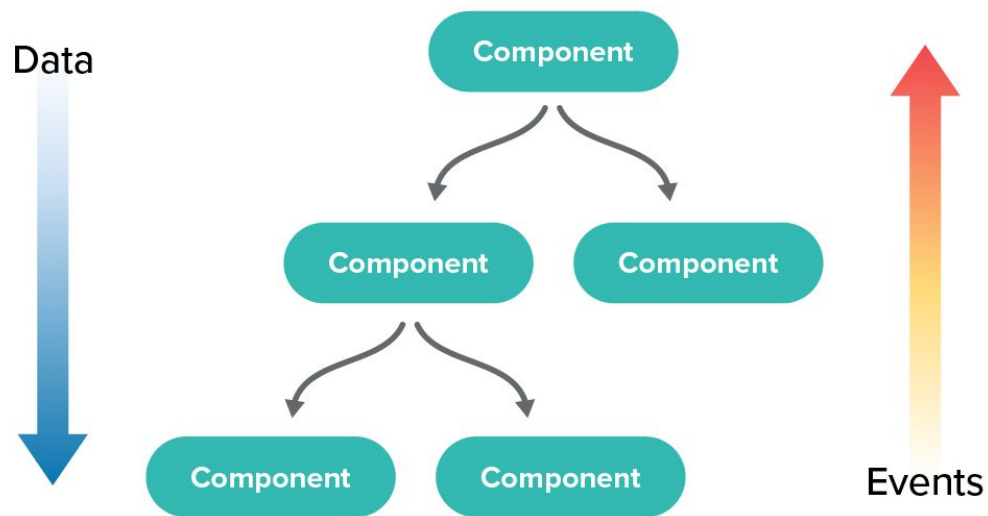
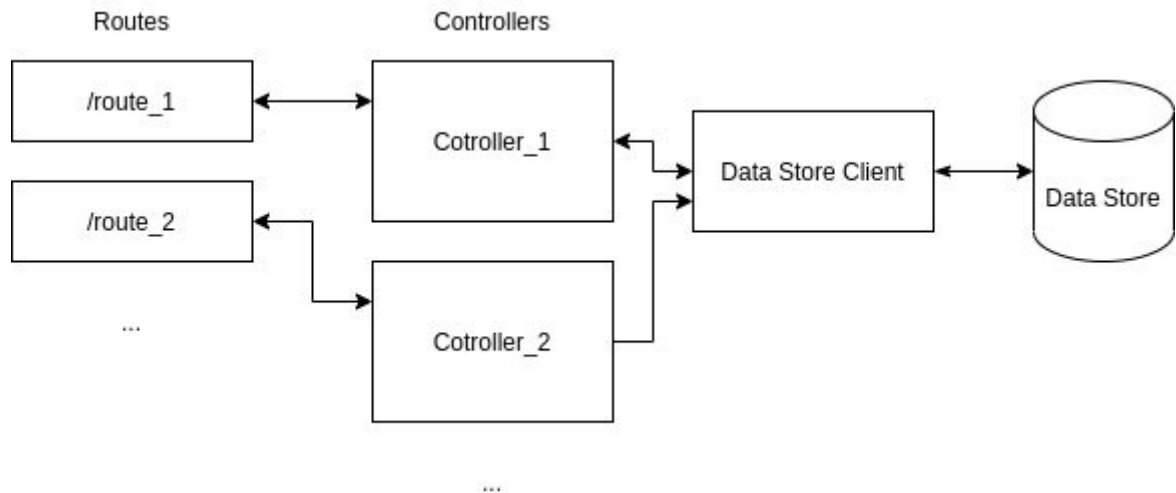


Figure 51: Component Architecture Logic

Each page belongs to a higher order component. The data is passed to the components below through props and the events are handlers that are passed from the parent components.

- *What is the back-end structure*

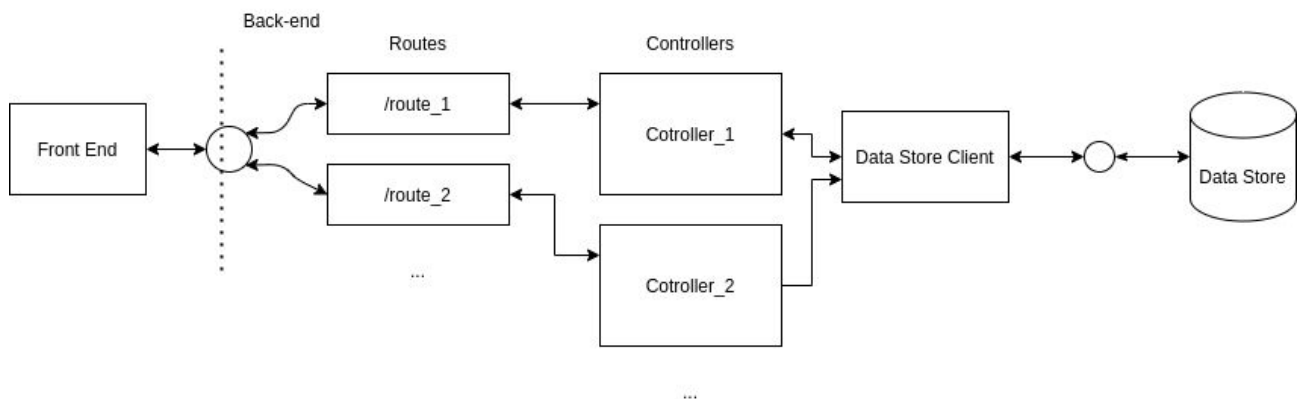
The back-end structure contains 4 base parts. The **routes** part that assigns specific actions to the routes that the client requests data from. The **controllers** part that are the actions that should be executed when a request arrives at the route. The **client** part the controllers use when needing to query the data store for data. And lastly the **data store part** that contains all of the data (surveys, answers, questions, etc) and executes queries which the client asks for.



Diagrams 38: Back-End Abstract Structure

- *How do the different entities communicate ?*

The front-end communicates with the back-end through API endpoints and calls upon them, in REST architecture environments. The same goes for the Data Store Client with the Data Store, they communicate through REST endpoints.



Diagrams 39: Back-End Structure Detailed

- *How is the data stored within the data store ?*

The questions, surveys and answered surveys are stored within our data store systems with a document-oriented format. Our database is non-relational and data is processed and aggregated through the Elasticsearch search-engine system that we will see in the technical analysis phase.

Now that we have covered the basic architectural decisions on the different implementation phases, we will move to the end of the Design Analysis chapter and explain the design principles that our system follows.

3.6.2 Design Principles

The design of our service aims to create a system that is:

- Scalable & Highly decoupled:

Developers can add a different set of question types, choose different visualization libraries or form question libraries, apply different themes and all that without messing or breaking the initial code structure and logic. The most widely used technique that we used to justify this is the conditional logic that is being used throughout the app. To remember it:

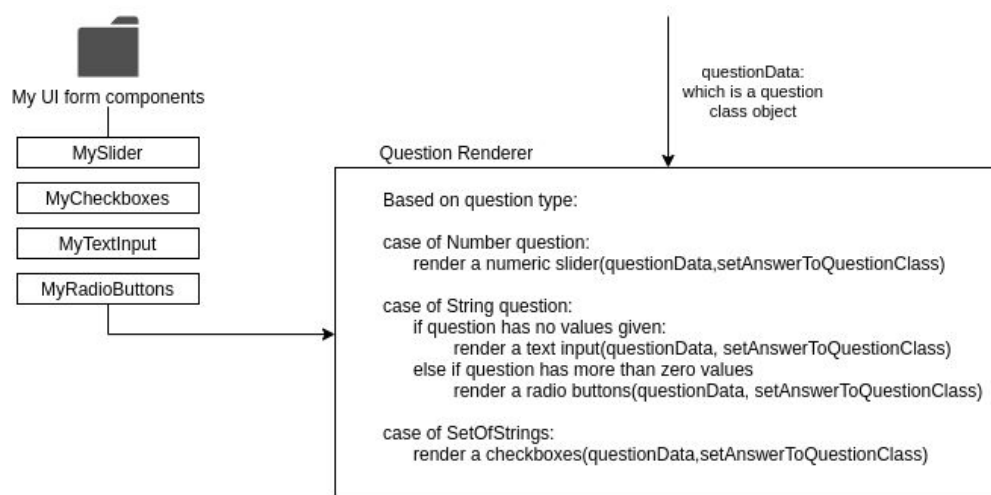


Diagram 40: Question Rendering Abstract

Each library rests in a different directory and code selects the selected UI on the run-time, allowing to remove or add libraries without changing the question rendering logic.

- Stable and Sturdy:

The front-end can be easily changed. The data store and its client can be changed with little effort. The individual parts can easily be detached, in a language-agnostic way, in order to produce timeless systems that are affected the least possible by the constant changes.

4. SYSTEM IMPLEMENTATION

In the previous chapter, that of Design Analysis, we focused on the design of the systems in a rather language-agnostic way. Although we saw many code examples, it is now time to proceed onto the technical explanation of each component, starting off with the technology stack which was used across our implementation.

4.1 The technology stack used

Once again, the whole platform abstract architecture is as follows :

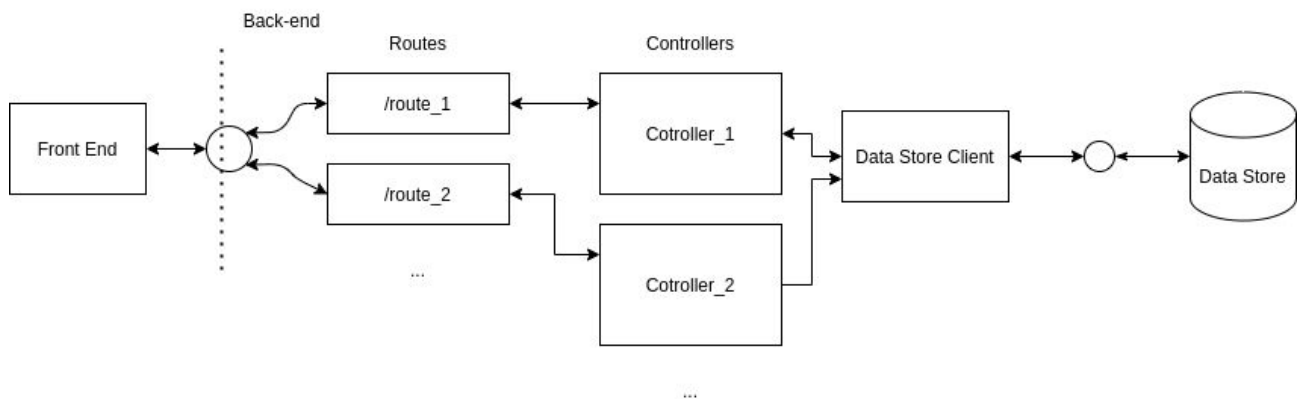


Diagram 41: Back-End Structure Detailed

In the front-end we used:

- The [React](#) javascript framework, along with the [Material-UI](#) for the UI building and [Recharts](#) for the results visualization.

In the back-end we used:

- The [Node.js](#) and the [ExpressJS](#) javascript framework for the server implementation along with the routing and controllers

And the data store is:

- [Elasticsearch](#) search engine, that stores the data and aggregates the results.

It also uses:

- [NPM](#) for the package management that our app uses
- [Babel](#) as the javascript compiler
- [ESLint](#) for the code formatting and optimization

Each of the above tools and frameworks will be explained in the following chapters.

The question that soon emerged before implementing this SaaS survey building platform was which technology stack was appropriate for our goal. We aimed for a full javascript implementation. The technology stack used consists of cutting edge technologies that most corporations use and have a rich community to resort to. We chose React for the front-end because it provides multiple libraries and it is the most popular js framework for web-development. NodeJS along with ExpressJs helps you set up a server easily in the same js manner and Elasticsearch is a very very powerful engine that has really fast indexing options and huge aggregation capabilities for the result analysis. Each of the above frameworks and services has its unique sense which we shall see.

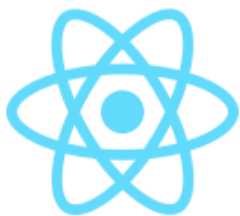
4.2 System Architecture

The stack will be explained starting from the front-end and then moving backwards to the back-end.

4.2.1 Front-End

The Front-End is fully implemented with the React javascript framework, but within we use a set of packages for the different functionality that is required and that we downloaded from the NPM package manager. In this chapter we will explain only the key packages that the front-end widely uses for its purposes, along with the core technologies.

4.2.1.1 React



[React](#) is a javascript library that creates component based user interfaces. Starting from simple and small components it can scale to complex and big components that have various functionalities. For the front-end implementation the version *16.13* was used. React allowed us to create all app components easily, by letting us divide the complex pages to its core parts, enhancing the code maintainability and scalability. More information about React can be

found on their site, that can be found on the references page.

4.2.1.2 Material UI

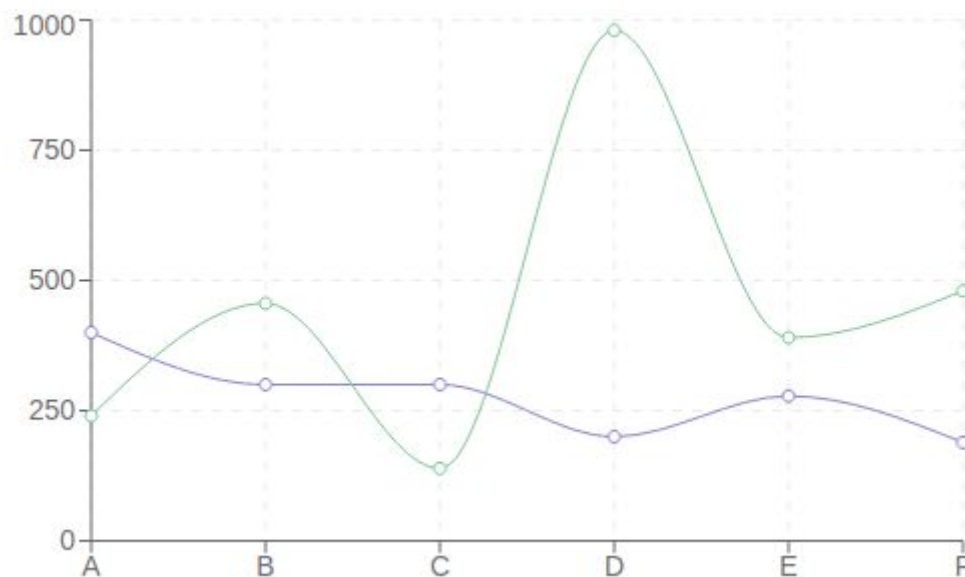


[Material UI](#) is a UI library for faster and easier web development in React. It provides a huge variety of aesthetic and minimal UI components. It has rich documentation, the components are highly customizable and it is the most popular UI react library, meaning it has a big community to help the programmers who make use of its services.

4.2.1.3 Recharts

```
<Recharts />
```

[Recharts](#) is a composable charting library built on React components, that offers a diversity of charts, graphs and plots while being widely customizable.



The whole implementation widely uses the above frameworks and libraries. The code can be found within our repos. In the chapters below we will showcase some programming techniques that React offers that had a significant impact on our implementation.

4.2.1.4 React Router

It is a collection of navigational components that compose declaratively with your application. Users get the content of pages by typing in their browsers urls. The React Router defines which component is going to be used for the different kinds of urls that exist. The routing that happens within the front-end is as follows :

```
<Router>
  <ScrollToTop>
    <Navbar changeTheme={this.switchTheme} />
    <Switch>
      <Route exact path="/" component={Home} />
      <Route exact path="/surveys" component={Surveys} />
      <Route exact path="/results" component={Results} />
      <Route
        path="/surveys/:survey_id/:section_idx"
        component={Survey}
      />
      <Route path="/about" component={About} />
      <Route path="/adminpage" component={AdminPage} />
      <Route path="/questionpool" component={QuestionPool} />
      <Route
        path="/editsurvey/:survey_id"
        component={EditSurvey}
      />
      <Route
        path="/editquestion/:question_id"
        component={EditQuestion}
      />
      <Route
        path="/results/:survey_id"
        component={ResultsPage}
      />
      <Route
        path="/visualization_wizard/:survey_id"
        component={VisualizationWizard}
      />
      <Route path="/conductors" component={Conductors} />
      <Route path="/terms" component={Terms} />
      <Route path="/working" component={UnderConstruction} />
      <Route component={NotFound} />
    </Switch>
    <Footer />
  </ScrollToTop>
</Router>
```

Code 22: React Router

This navigates the user to the right page, according to what they ask for, while also giving the option to add URL parameters for the components that will get rendered. Also, if the router does not find a URL that corresponds to a component, then it will render the NotFound page.

4.2.1.5 Local Storage

The platform offers a way for users to store some data in the browsers local and session storage. Options like remembering user preferences about light or dark themes or continuing a survey if the user closed the window by mistake improves the user experience and adds many conveniences to the process.

For example, if a user has completed more than 10% of the survey and by mistake closes the window, then when returning to the survey page they will get a message asking them if they want to continue the survey.

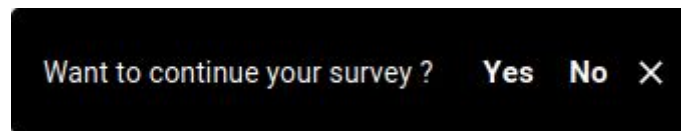


Figure 52: Continue Survey

And the code that uses the local storage is as follows :

```

componentWillUnmount() {
  // before exiting the survey, react will store the survey to local storage
  this.storeSurveyToLocalStorage();
}

storeSurveyToLocalStorage = () => {
  // store survey & its completion rate to local storage
  localStorage.setItem("survey", JSON.stringify(this.surveyCreated));
  localStorage.setItem(
    "completionRate",
    this.surveyCreated.getCompletionRate()
  );
  window.removeEventListener(
    "beforeunload",
    this.storeSurveyToLocalStorage
  );
};

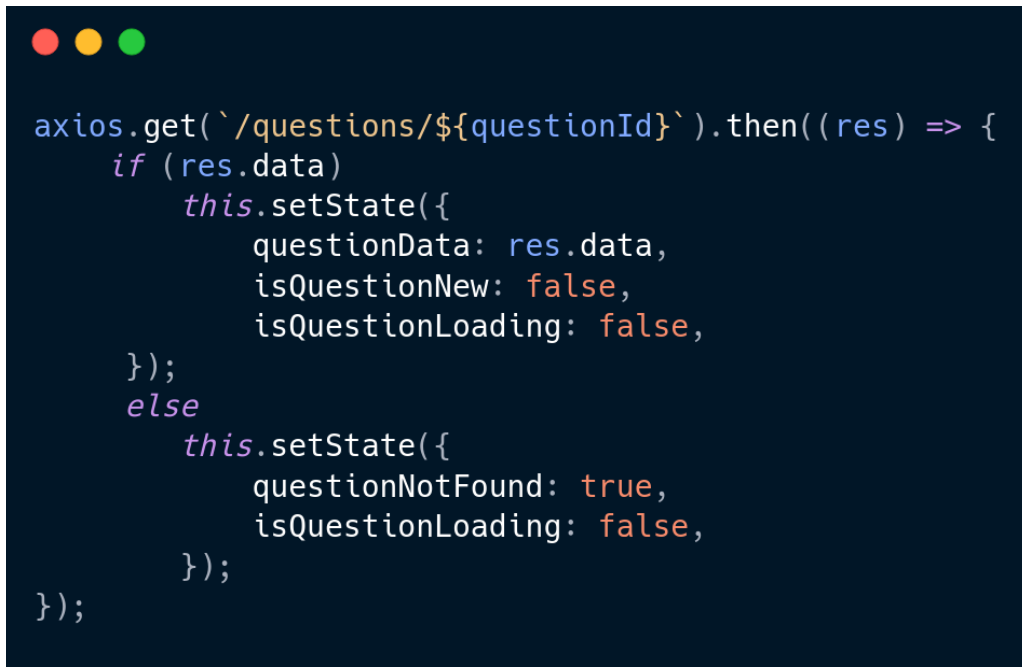
```

Code 23: Continue Survey

Every time a survey is going to be closed from the screen, then the survey is stored to local storage. And everytime the user goes to take the survey again and wants to continue the survey, the Survey Class Method populateSurveyForm will be invoked with the parameter the stored survey which resides in the local storage.

4.2.1.6 Axios and Data Fetching

Front-end constantly gets data from the back-end for the different needs. The data is fetched from the backend endpoints with the help of the [Axios](#) HTTP client. An example usage that is invoked whenever a user wants to edit a question:



```

axios.get(`/questions/${questionId}`).then((res) => {
  if (res.data)
    this.setState({
      questionData: res.data,
      isQuestionNew: false,
      isQuestionLoading: false,
    });
  else
    this.setState({
      questionNotFound: true,
      isQuestionLoading: false,
    });
});

```

Code 24: Axios Data Fetching

4.2.1.7 React Context

In case there are multiple nested components and the component tree goes into a big depth, it is wise to have a way for the component-tree leaf nodes to communicate with the root. The React's usual way to pass data to children as props, but in this case these props would need to traverse multiple levels, making the code harder to maintain and scale. This is where React Context Provider comes into play. It provides a way for data and methods to be accessible wherever in the component tree. A real use-case of the React Context Provider is shown in the example below, where the Questions Selected Data needs to get the aggregation parameters for a specific question from the rendered question parameters forms.

The problem that we faced when building the Questions Selected Data was that the Visualization Wizard should somehow be able to get the aggregation parameters data whenever those changed. The Visualization Wizard will use the React Context in order to provide the Question Parameters Forms the method that will call whenever the user

changes the aggregation filters. This method is just an update Visualization Wizard Questions selected data.

The method is as follows :

```
updateQuestionsAggrData = (questionAggrData, questionIndex) => {
  const updatedQuestionsAggrData = [...this.state.questionsAggrData];
  updatedQuestionsAggrData[questionIndex].q_args = questionAggrData;
  this.setState({ questionAggrData: updatedQuestionsAggrData });
};
```

Code 25: Update Questions Aggregation Data Method

And the whole context providing logic is showcased below:

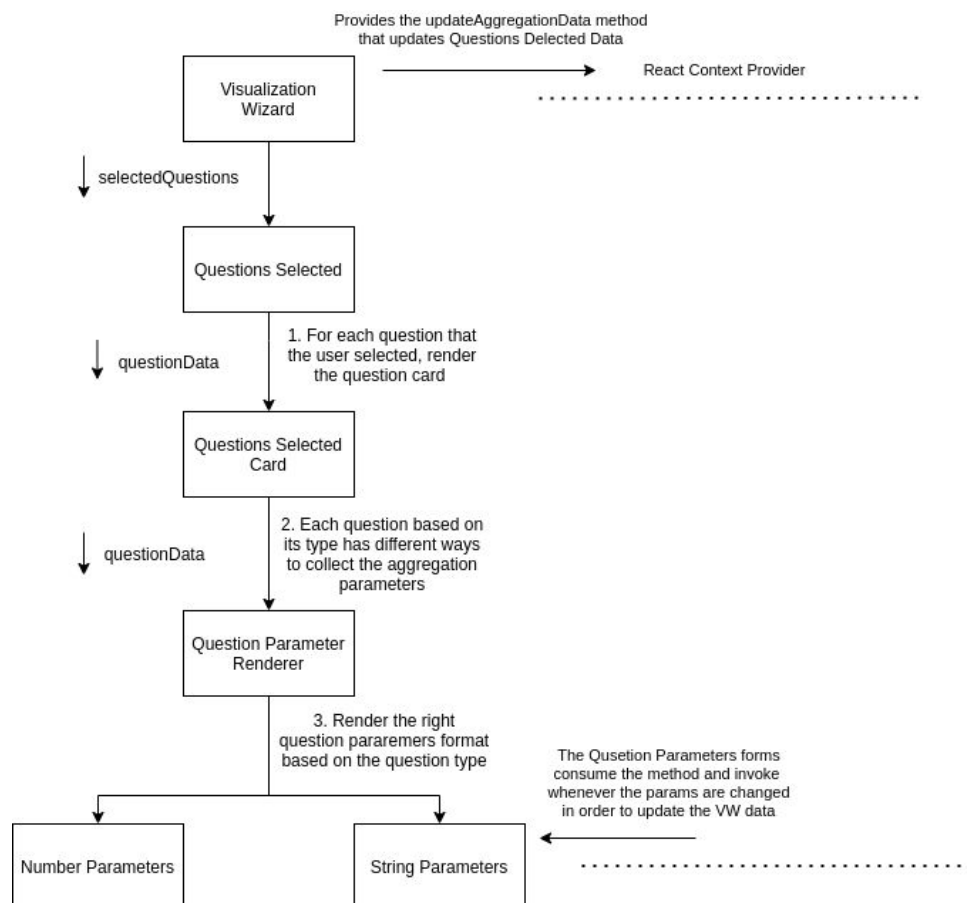
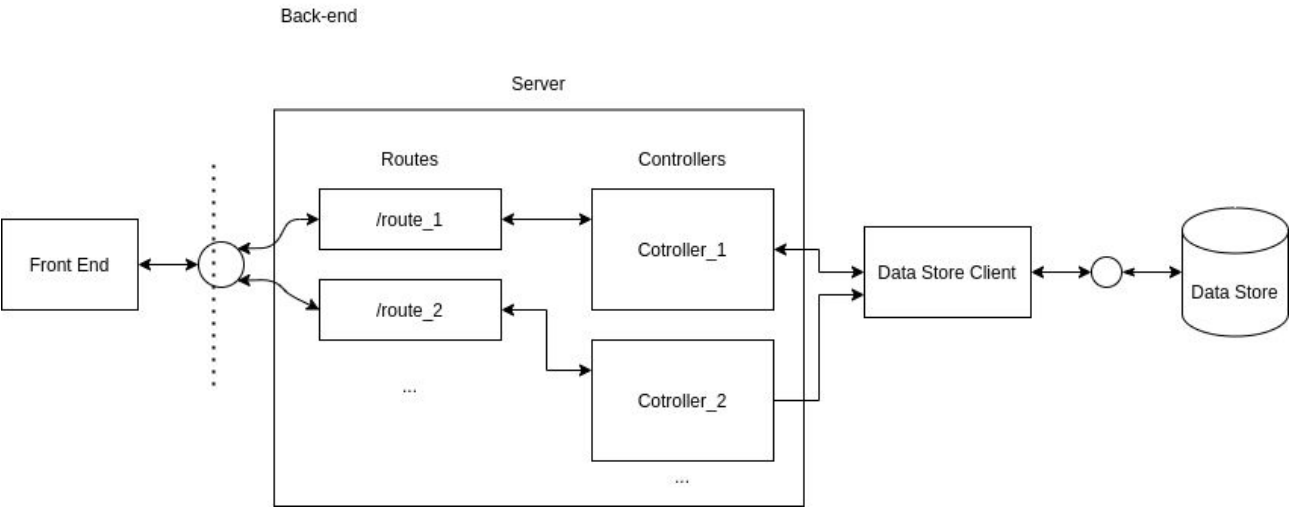


Diagram 42: React Context

4.2.2 Back-End

The backend consists of the servers that contain the routes and the controllers, the data store client and the data store.

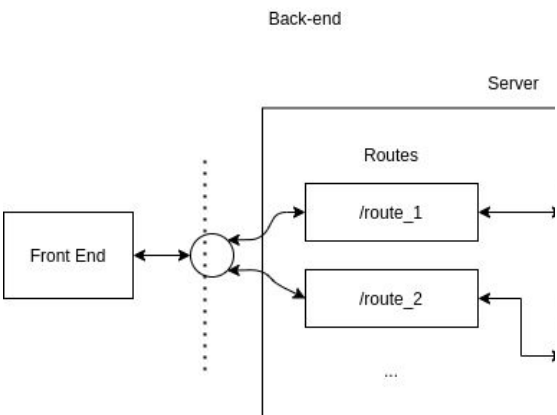


Code 25: Back-End Abstract Strute

Before proceeding to explain one by one the technologies that are used in the back-end, we will first explain the REST APIs that the back-end handles.

4.2.2.1 Rest API & Endpoints

The implementation is a RESTful application, meaning it uses the [REST](#) (Representational State Transfer) architecture style for the front-end and back-end in-between communication.



Code 25: Update Questions Aggregation Data Method

Here is a table with all the possible rest endpoints that clients can use and what the server does when a client requests from them.

API Route	API Method	Server Behavior
/questions/:sortBy&:order	GET	Returns all the questions on the order specified
/questions	POST	Adds a question on the data store
/questions/new_id	GET	Get a new id for a newly created question
/questions/search	GET	Return a number of questions based on a search term
/questions/:qid	GET	Return the question specified by the route parameter
/questions/:qid	PUT	Update the question specified by the route parameter
/questions/:qid	DELETE	Delete the question specified by the route parameter
/surveys/	GET	Gets all the surveys from the data-store
/surveys/questions_existis_in/:qid	GET	Get all the surveys that the question with the specific id resides in
/surveys	POST	Adds a survey on the datas-store
/surveys/:sid	GET	Get the survey with the specified id from the data-store
/surveys/:sid	PUT	Update a survey from the data-store
/surveys/:sid	DELETE	Deletes a survey from the data-store
/answered_surveys/:sid	GET	Get all the answered surveys of a specific survey
/answered_surveys	POST	Add an answered survey to the data-store
/aswered_surveys/question_stats	GET	Get the statistics a about a question based on the answered surveys
/answered_surveys/survey_stats	GET	Get survey statistics about its answers
/answered_surveys/wizard_stats	GET	Get wizard statistics for the Visualization Wizard

Table 1: REST API Endpoints

4.2.2.2 ExpressJS

Express



[Express](#) is a minimal and flexible [Node.js](#) web application framework that provides a robust set of features for web and mobile applications. We used Express to create our server:

```
// Import express framework
const express = require("express");

// Import middleware
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
const compression = require("compression");
const helmet = require("helmet");
const cors = require("cors");

// Import routes
const questionsRouter = require("./routes/questions-route");
const surveysRouter = require("./routes/surveys-route");
const answered_surveysRouter = require("./routes/answered_surveys-route");

// Setup default port
const PORT = process.env.PORT || 4000;

// Create express app
const app = express();

// For every request display details
app.use(function (req, res, next) {
  var now = new Date();
  console.log(
    `\n\n\n[${now.toUTCString()}] Express received a ${
      req.method
    } request on ${req.originalUrl} :`
  );
  next();
});

// Implement middleware
app.use(cors());
app.use(helmet());
app.use(compression());
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(bodyParser.json());

if (process.env.NODE_ENV && process.env.NODE_ENV !== "development") {
  app.get("*", (req, res) => {
    res.sendFile("build/index.html", { root: __dirname });
  });
}

// Implement route for '/questions' endpoint
app.use("/questions/", questionsRouter);

// Implement route for '/surveys' endpoint
app.use("/surveys/", surveysRouter);

// Implement route for '/answered_surveys' endpoint
app.use("/answered_surveys/", answered_surveysRouter);

// Implement route for errors
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send("Something broke!");
});
```

Code 26: Server Implementation

The express implements the back-end logic serving the routes that we described with the corresponding controllers.

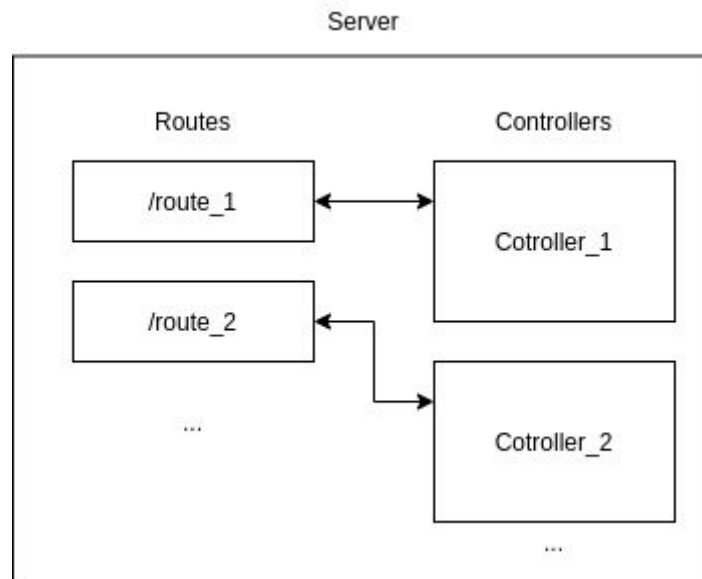


Diagram 43: Routes And Controllers

Following are some examples regarding each implementation:

i) Routes

Routes are endpoints that the backend serves. Let us see how the surveys routing works with a code example:

```

// Import express
const express = require("express");

// Import home controller
const answered_surveysController = require("../controllers/answered_surveys-controller.js");

// Create express router
const router = express.Router();

/* Create a route between controller and API Methods */

// GET the answered surveys
router.get("/", answered_surveysController.getAnsweredSurveys);

// POST an answered_survey to elasticsurveys
router.post("/", answered_surveysController.postAnsweredSurvey);

// GET question stats from the answered survey
router.get("/question_stats", answered_surveysController.getQuestionStats);

// GET answered surveys stats
router.get("/survey_stats", answered_surveysController.getSurveyStats);

// GET visualization wizard results
router.get("/wizard_stats", answered_surveysController.getWizardStats);

// Export router
module.exports = router;

```

Code 27: Routes

Each route is binded with a specific controller which contains the logic for each route.

ii) Controllers

A controllers example is as follows :

```
// Import elasticsearch
var elastic = require("../elasticsearchClient");
const util = require("util");
const { response } = require("express");
// Set the index name for the controller
const indexName = "answered_surveys";

// Controller for GET request to '/answered_surveys'
exports.getAnsweredSurveys = async (req, res) => { ...
}

// Create controller for POST request to '/answered_surveys'
exports.postAnsweredSurvey = async (req, res) => {
  console.log("→ [postAnsweredSurvey] controller will handle the request");
  elastic
    .postDocumentToIndex(indexName, req.body, makeid(10))
    .then((result) => {
      let response_data = {
        status: null,
        message: null,
      };
      if (result.result === "created") {
        response_data.status = "success";
        response_data.message = "Successfully posted an answered survey ";
      } else {
        response_data.status = "error";
        response_data.message =
          "There was an error in posting the answered survey ";
      }
      res.send(response_data);
    });
});

// Create a controller for GET request to '/answered_surveys/question_stats?
exports.getQuestionStats = async (req, res) => { ...
};

// Create a controller for GET request to '/answered_surveys/survey_stats?
exports.getSurveyStats = async (req, res) => { ...
};

Object.byString = function (o, s) { ...
};

// Create a controller for GET request to '/answered_surveys/survey_stats?
exports.getWizardStats = async (req, res) => { ...
};
```

Code 28: Controllers p.128

It actually contains all the code logic that is required in order for the system to operate with the data-store for the desirable action, like getting the survey with a specific id etc.

With ExpressJS setting up the server and the routing/controlling of the back-end we move towards the final part. The Elasticsearch's data store and client, along with details about it.

4.2.2.3 Elasticsearch



[Elasticsearch](#) is a distributed, open source search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. Elasticsearch is built on Apache Lucene. Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the central component of the Elastic Stack, a set of open source tools for data ingestion, enrichment, storage, analysis, and visualization

We use Elasticsearch for the data-storing, the full-text search capabilities and the data ingestion capabilities.

I. Index Organization

An **Elasticsearch index** is a collection of documents that are related to each other. **Elasticsearch** stores data as JSON documents. To store the essential data for our systems we used **3** indexes. One for the *questions*, one for the *answers* and one for the *answered* surveys. The documents that are stored within each index follow the same data format that we explained in the Data Analysis phase.

II. Queries

In order to operate with Elasticsearch and get, post, update, delete data from it, apply a set of aggregations and more, we need a querying system. Elasticsearch provides a rich set of queries that users can utilize and a domain specific language that creates them, the [Query DSL](#).

III. Elasticsearch JS Client

[Elasticsearch JS Client](#) is our data-store client that executes the queries on the elasticsearch indexes. It is the intermediate node between the express controllers and elasticsearch indexes and the direct communicator with the data-base.

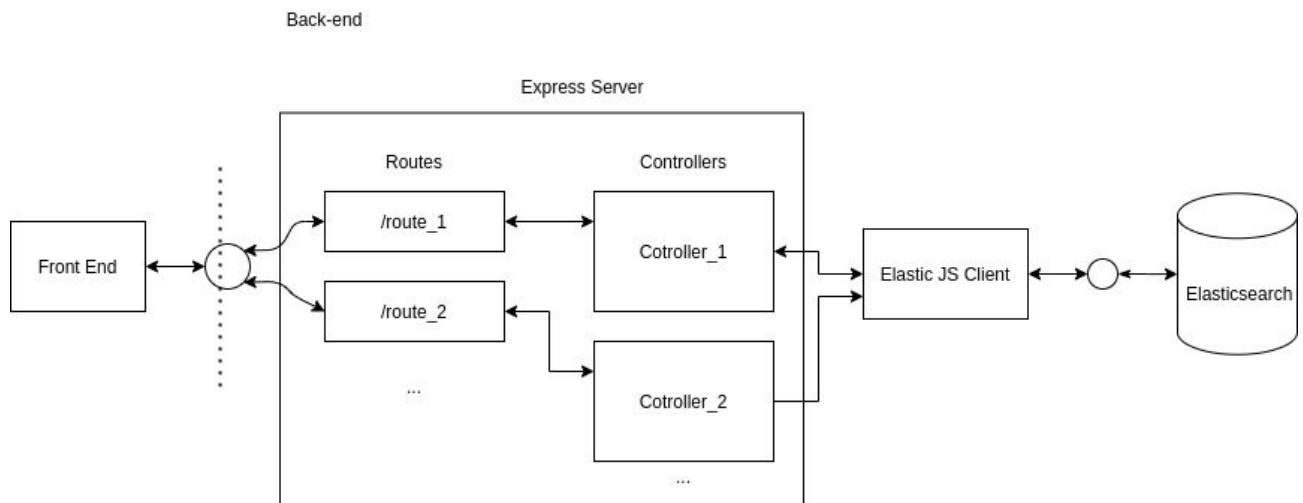


Diagram 44: Elasticsearch JS Client

The controllers use the Elastic JS Client for example as follows:

```

// Create controller for GET request to '/questions/:sortBy&:order'
exports.getQuestions = async (req, res) => {
  console.log("→ [getQuestions] controller will handle the request");
  elastic
    .getDocumentsFromIndex(
      "questions",
      50,
      req.params.sortBy,
      req.params.order
    )
    .then((result) => {
      var questions = [];
      result.hits.hits.forEach((hit) => {
        questions.push(hit._source);
      });
      res.send(questions);
    });
};

```

Code 29: Elastic JS Client getQuestions Method

Here the getQuestions controller invokes the asynchronous getDocumentsFromIndex method that resides within the elasticClient. Upon response, it gets the questions and forwards them towards the client.

All the functions that operate directly with the Elasticsearch are defined within the elasticsearchClient file. Here is the code example of the elastic JS Client that contains

those functions. Notice how the elasticClient uses the Query DSL in order to operate with Elasticsearch's data store.

```
var elasticsearch = require("elasticsearch");
const { search } = require("../routes/surveys-route"); // ?

// Initialize the elasticsearch js client
var elasticClient = new elasticsearch.Client({
  host: "localhost:9200",
  log: "info",
});

// Get a number of documents from the an elasticsearch's index
async function getDocumentsFromIndex(
  indexName,
  numOfDocs = 200,
  sortBy,
  order
) {
  console.log("→ [getDocumentsFromIndex] will query elasticsearch");
  return elasticClient
    .search({
      index: indexName,
      sort: `${sortBy}.keyword:${order}`,
      body: {
        size: numOfDocs,
        query: { match_all: {} },
      },
    })
    .then(
      function (elasticsearch_response) {
        console.log("\t... ✓ Successful operation with elasticsearch");
        return elasticsearch_response;
      },
      function (err) {
        console.log(
          "\t... ✗ Not-successful operation with elasticsearch: "
        );
        console.trace(err.message);
        return err.message;
      }
    );
}

exports.getDocumentsFromIndex = getDocumentsFromIndex;

// Get a document from elasticsearch index
async function getDocumentFromIndex(indexName, docID) { ...
}
exports.getDocumentFromIndex = getDocumentFromIndex;

// Post a document in elasticsearch's index
async function postDocumentToIndex(indexName, document, docID) { ...
}
exports.postDocumentToIndex = postDocumentToIndex;
```

Code 30: Elastic JS Client Methods

IV. Kibana



[Kibana](#) is a free and open user interface that lets you visualize your Elasticsearch data and navigate the Elastic Stack. Do anything from tracking query load to understanding the way requests flow through your apps. In our app we used Kibana to query the indexes from its console and visualize the results, too.

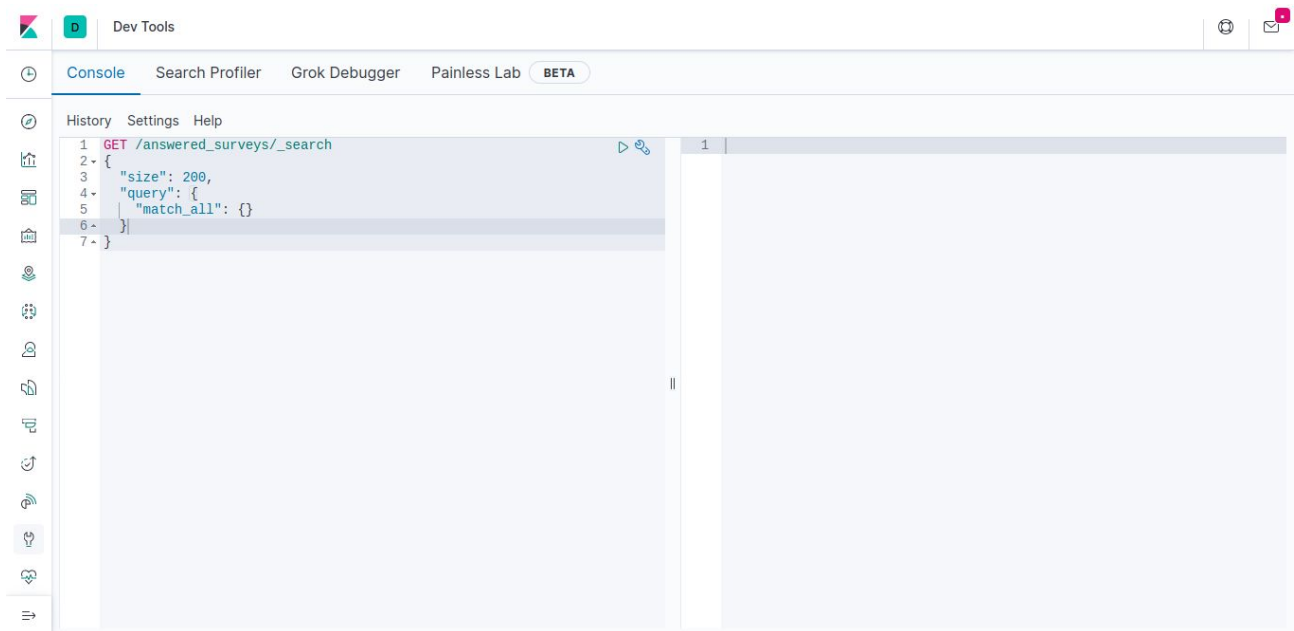


Figure 52: Kibana

V. How Aggregations Work

For the last part of the elasticsearch explanation analysis, it is wise to understand at least in an intuitive level how the aggregations work within elasticsearch.

Each index in elasticsearch has a specific mapping, which explains to elasticsearch what is the document that stores structure, along with a set of fields containing metadata about each field.

Elasticsearch does this to make the searching and aggregation capabilities as optimized and performant as possible. We won't analyze the mappings here, more info can be found [here](#).

For the time being, let's remember how the elasticsearch stores the answered surveys within its *answered_surveys* index.

```

{
  "id" : "STUD",
  "took" : 4,
  "submittedOn" : "9/10/2020",
  "answers" : {
    "Q01" : [
      "Οικονομικά Κίνητρα",
      "Εξωτερικές Πιέσεις"
    ],
    "Q02" : "Ναι",
    "Q03" : [
      "Οι γνώσεις μου δεν είναι αρκετές για την αγορά εργασίας",
      "Δεν ξέρω ακριβώς τον τομέα πληροφορικής που θέλω να ασχοληθώ"
    ],
    "Q04" : "Όχι",
    "Q05" : [
      "Καλύτερη δικτύωση",
      "Περισσότερη φυσική παρουσία"
    ],
    "...",
    "Q28" : "Ναι",
    "Q29" : 8,
    "Q30" : 7,
    "Q31" : "Όχι",
    "Q32" : "Λίγο",
    "Q33" : "Τμήμα Πληροφορικής & Τηλεπικοινωνιών ΕΚΠΑ",
    "Q34" : "6",
    "Q35" : "7.6",
    "Q36" : "Άντρας"
  }
}

```

Each answered survey has a field called answers, that contains the answer for each question. The key is the question id and the value of the answer or the set of answers for the specific question. Given the query below let's see how elasticsearch innate logic will handle the request.

```

{
  size: 0,
  query: {
    bool: {
      must: [ { match: { id: 'STUD' } } ],
      should: []
    }
  },
  aggs: {
    agg_0: {
      terms: { field: 'answers.Q02.keyword' },
      aggs: {
        agg_1: {
          terms: { field: 'answers.Q11.keyword' },
          aggs: { agg_2: { stats: { field: 'answers.Q29' } } }
        }
      }
    }
  }
}

```

Elasticsearch stores within its `answered_survey` index a set of `answered_surveys` with the format above.

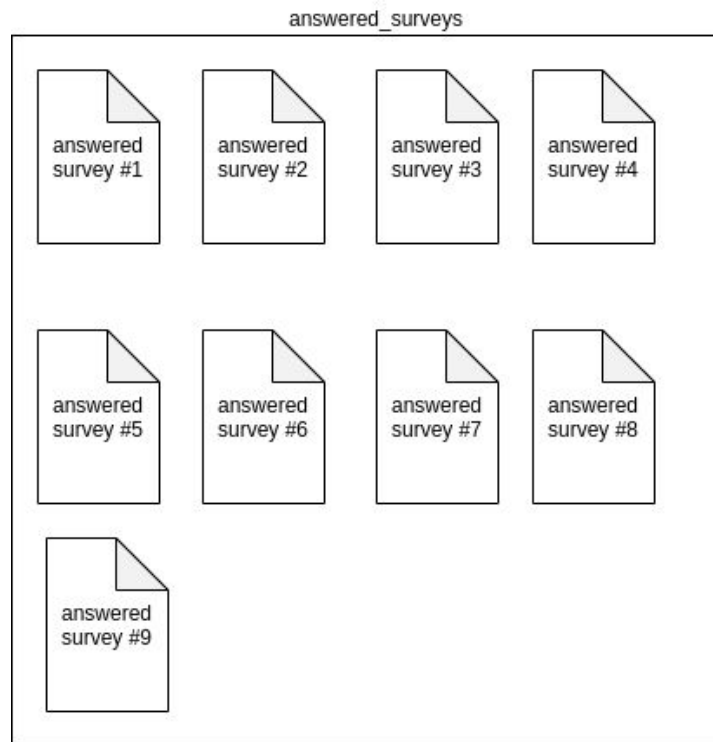


Diagram 45: Documents In Index Without Aggregations

The first thing that elastic will do is to apply a boolean query in order to match all the answered surveys that must be of id 'STUD'.

```
bool: {
  must: [ { match: { id: 'STUD' } } ],
```

The above part of the query will yield all the STUD surveys.

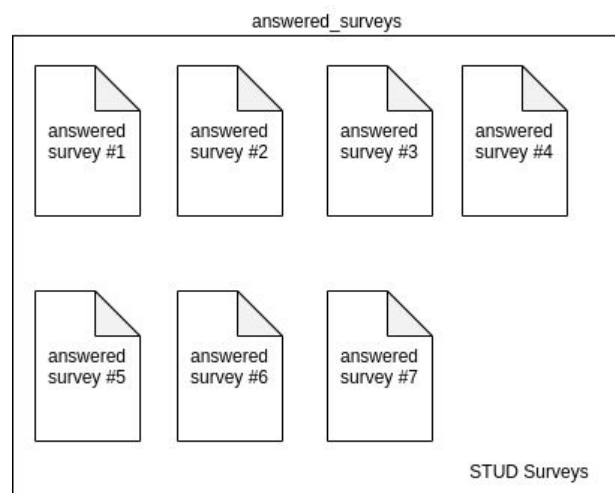


Diagram 46: Survey ID Match Aggregation

Then for all the STUD surveys, because we haven't other queries, elasticsearch will start the aggregations.

```

aggs: {
  agg_0: {
    terms: { field: 'answers.Q02.keyword' },
    aggs: {
      agg_1: {
        terms: { field: 'answers.Q11.keyword' },
        aggs: { agg_2: { stats: { field: 'answers.Q29' } } }
      }
    }
  }
}

```

The first aggregation is the *agg_0* which is a terms aggregation that happens based on the Q02 answers. The *keyword* suffix in the aggregations means the answers of the Q02 question should be treated as keyword that the aggregations will happen upon. The possible answers of the Q02 are “Yes” and “No”. Based on these answers, elasticsearch creates two buckets. The one contains the answered surveys that had “Yes” answers to Q02 and the other the answered surveys that had “No” answers to Q02.

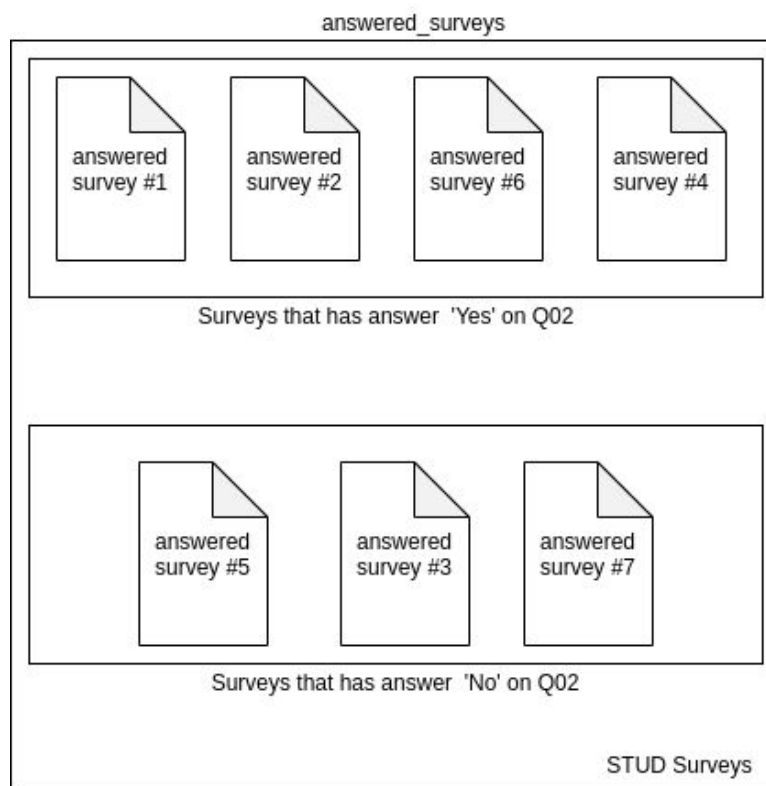


Diagram 47: Documents After Aggregation 0

After the aggregation 0, the elasticsearch proceeds to apply the second aggregation to each bucket.

```

agg_1: {
  terms: { field: 'answers.Q11.keyword' },
  aggs: { agg_2: { stats: { field: 'answers.Q29' } } }
}

```

The *agg_1* now splits the answers based on the Q11 answers, that again are “Yes” or “No”. This aggregation happens on each bucket that the *agg_0* created, producing the following result:

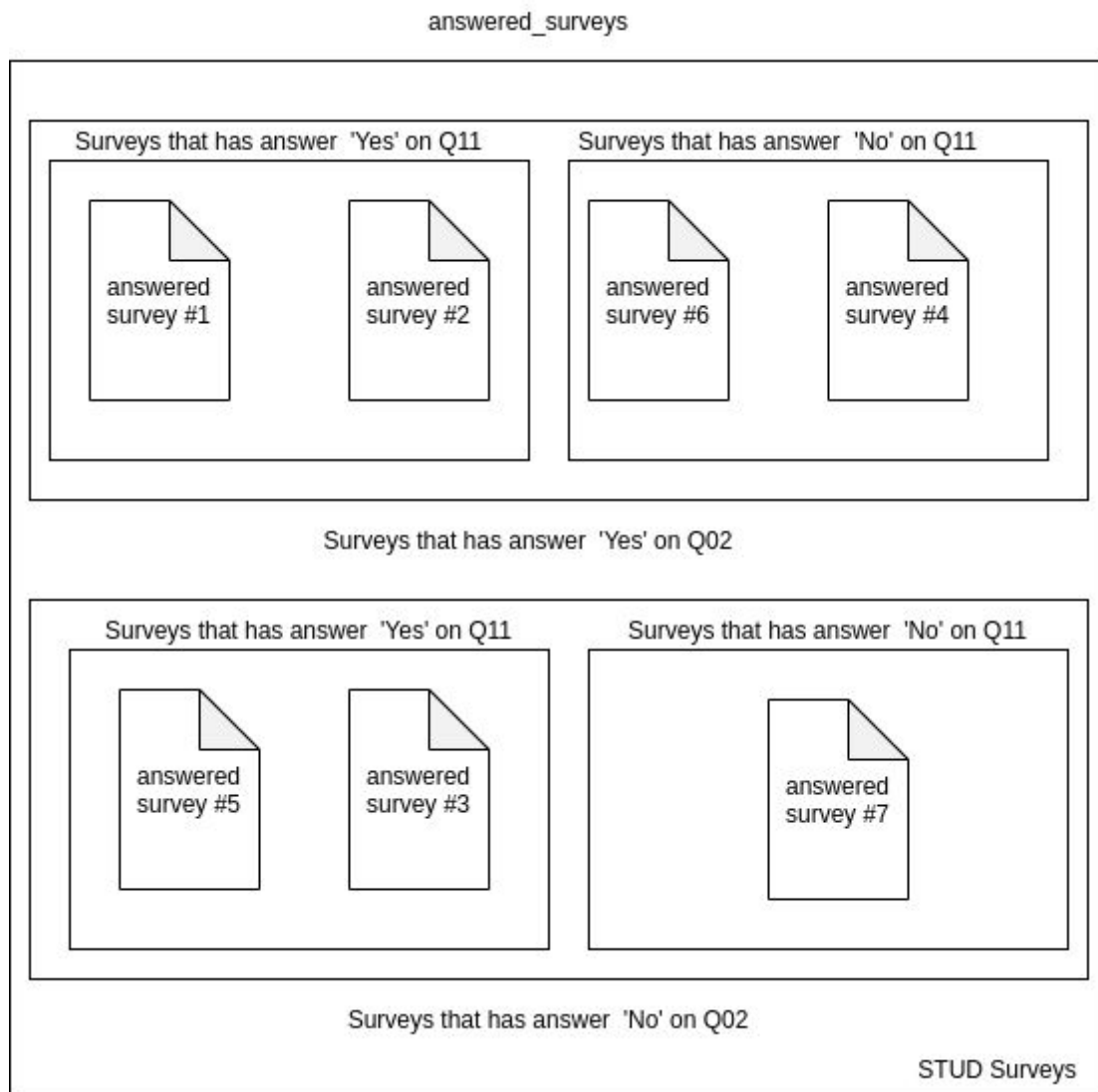


Diagram 48: Documents After Aggregation 1

Given now the four distinct nested buckets, the last aggregation is applied.

```
aggs: { agg_2: { stats: { field: 'answers.Q29' } } }
```

This aggregation is a stats aggregation. Based on the answer of the Q29 question, the elasticsearch for the documents that exist in the buckets above, does the stats aggregation that computes the minimum, maximum, avg, count and other information for the Q29 question answers that the buckets contains.

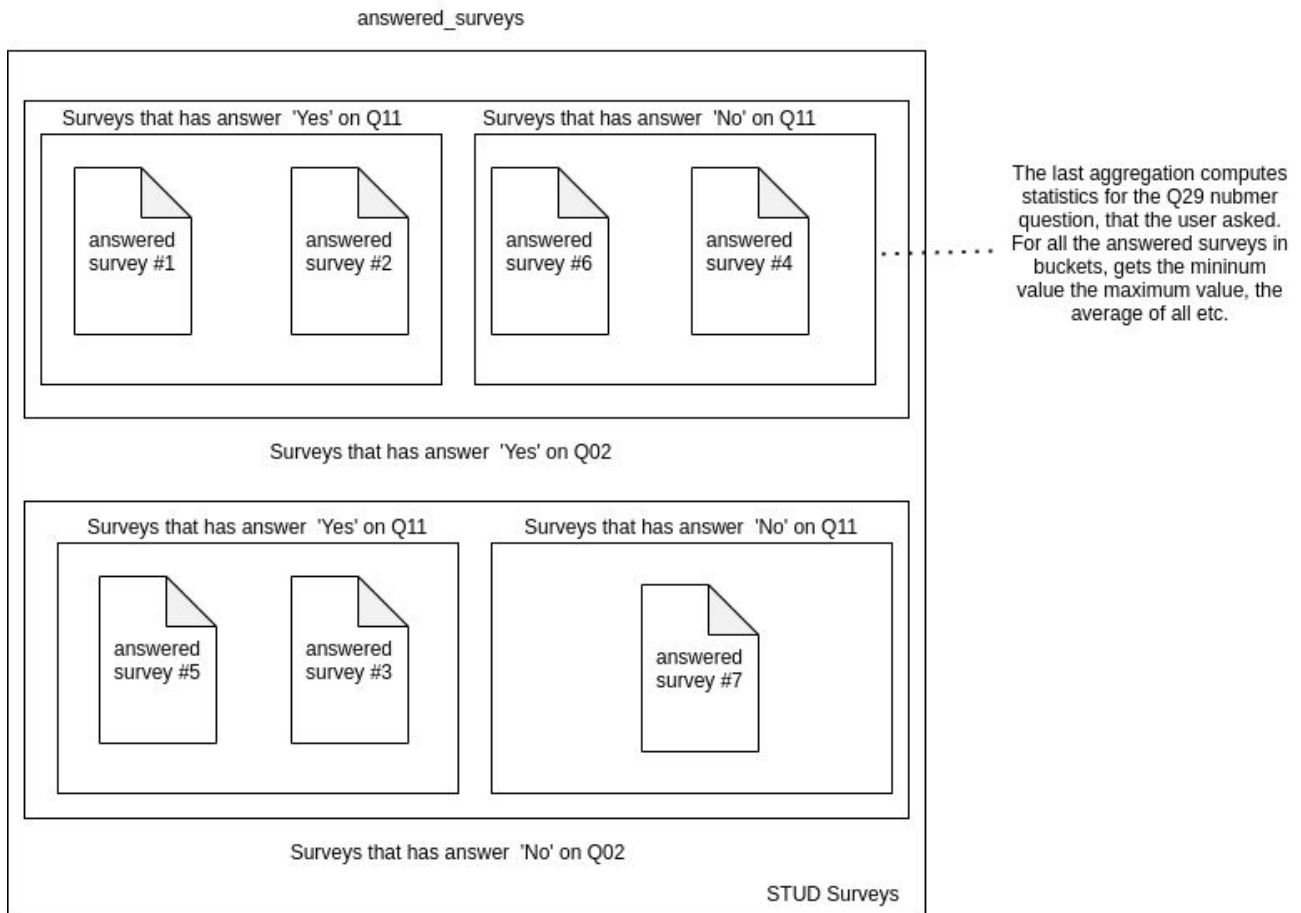


Diagram 49: Documents In Index Last Aggregation

The above aggregation yields its response with the aggregation info of each bucket that later the visualization wizard renders, with the procedure that we have explained in the 2.4.2 Chapter.

VI. Full-Text Search

The front-end utilizes the elasticsearch capabilities, providing the user full text experience. In a full-text search, a [search engine](#) examines all of the words in every stored document as it tries to match search criteria (for example, text specified by a user). We use Elasticsearch for example in the Question Pool, in order to easily search surveys. The showcase of the Full-Text Search along with the code implementation follows:









Question Pool			
Manage & Organize your questions			
<input type="text" value="Q Q1"/>			
Id	Type	Text	Actions
Q01	SetOfStrings	Πατί σκοπεύετε να ασχοληθείτε επαγγελματικά με την ανάπτυξη λογισμικού;	 
Q11	String	Έχετε αναπτύξει δικό σας λογισμικό (side-projects);	 
Q12	String	Έχετε ασχοληθεί με την συγγραφή κώδικα σε Open source;	 
Q13	Number	Θεωρώ τον εαυτό μου καλύτερο προγραμματιστή από το παρακάτω ποσοστό των φοιτητών μου.	 

Figure 53: Full-Text Search

In the above example the user typed Q1 and the server responded with all the questions that their id contains the 1 number. The search terms are applied to any field, applying full text search experience. The full-text search is provided by our back-end, with an API call to the `/questions/search` endpoint. The controller of that route executes the `ElasticClient` that does a search query on the Elasticsearch that yields all the questions.

```
// Get the documents by full-text searching an index
async function fullTextSearchDocument(indexName, searchTerm) {
  console.log("→ [fullTextSearchDocument] will query elasticsearch");
  return elasticClient
    .search({
      index: indexName,
      body: {
        query: {
          multi_match: {
            fields: ["*"],
            query: searchTerm,
            fuzziness: 2,
          },
        },
      },
    })
    .then(
      function (elasticsearch_response) {
        console.log("\t... ✓ Successful operation with elasticsearch");
        return elasticsearch_response;
      },
      function (err) {
        console.log(
          "\t... ✗ Not-successful operation with elasticsearch: "
        );
        console.trace(err.message);
        return err.message;
      }
    );
}
```

Code 31: Full-Text Search

4.3 Summing Up

Summing up the system architecture and the technology stack used to implement it, our code base is a Client-Server Restful architecture. For the client-side or Front-End we use React, Material-UI and Recharts frameworks and libraries for the UI building and graph plotting. For the Back-End or server we use the NodeJS and Express framework that builds the routes and controllers for the two ends to communicate. Our data-storing and processing happens with the Elasticsearch search engine and its client that let javascript to query its indexes.

An abstract diagram of the whole technology stack is the following:

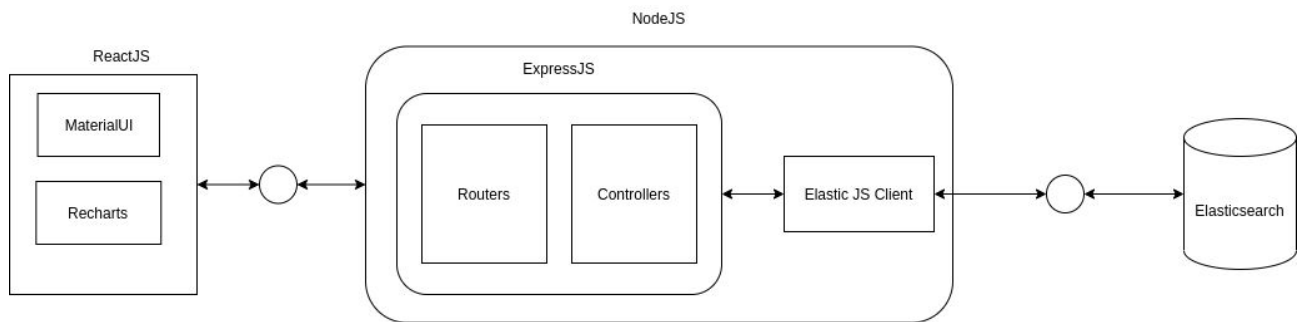


Diagram 50: Full-Stack Abstract Architecture

4.4 Technical Challenges

During the design phase of implementation we were searching for a framework or tool in order to build our backend. I came across a new backend as a service tool called [Firebase](#). It offered a lot of conventions and provided an easy way to manage the data stored within it, like a console and variety of plugins and automations. So even before analyzing what our project requirements and goals were I rushed to pick that data store as a service, thinking it was the perfect solution for the problem, despite the fact that it was a hurried decision. The end result was 1 month of coding and struggling to understand a tool that was far from suitable for a backend with so many demands. It did not have any options to aggregate so much data, neither was it so performant. So, I had to remove the whole backend and start all over. Of course my supervisor guided me by informing me that there were better alternatives, so we decided on the right tools, that being Express and Elasticsearch, and we built the back-end that is today and is perfect for the given problem.

Of course, it was not that Firebase was a bad tool. It just was not the right tool for the job. What I understood was that architectural decisions can cost a lot of time and in cases money and they do not change easily. So, never rush when it comes to the designing phase. Explore all the possible solutions, understand the problem and its requirements and after doing your best to brainstorm all the possible cases, then select the right tool and make sure that it will work.

5. CONCLUSION

The current state of the implementation is a fully functional survey manager, that offers users the ability to create and conduct surveys to the public, while also providing powerful mechanisms to get reports about the results and explore the possible correlations between questions and data with deep-nested visualizations.

For the future, the implementation will be enriched with more questions and visualization types and capabilities. Also, the platform will be thoroughly tested in various testing environments, optimized and get deployed in order to be ready to conduct and handle a series of surveys.

All in all, the whole thesis experience matured me as a person and gave me the chance to have a more well-rounded programming sophistication. It helped me sharpen a handful of programming skills and better prepare me for the future, for the best to come. And I am grateful to my university for giving me that opportunity.

TABLE OF TERMINOLOGY

Ξενόγλωσσος Όρος	Ελληνικός Όρος
Back End	Πίσω Μέρος
Front End	Μπροστά Μέρος
Deployment	Ανάπτυξη
Framework	Εργαλεία / Σκελετός Ανάπτυξης
Cloud	Υπολογιστικό Νέφος

ABBREVIATIONS

SaaS	Software as a Service
REST	Representational State TRansfer
UML	Unified Modeling Language
JSON	JavaScript Object Notation
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
API	Application Programming Interface
HTTP	HyperText Transfer Protocol

REFERENCES

- [1] Hackers & Painters, Paul Graham, May 2003 (<http://www.paulgraham.com/hp.html>) [Accessed 14/11/20]
- [2] Worse is better, P. R. Gabriel, 1994 *The Unix-Haters Handbook*, part of the handbook (https://cs.stanford.edu/people/eroberts/courses/cs181/projects/2010-11/WorselsBetter/index43bb.html?title=Main_Page&oldid=86) [Accessed 17/11/20]
- [3] Knuth Turing Award Speech, 1974 recipient of the [ACM Turing Award](#) [Accessed 9/11/20]
- [4] SE education SPLASH 2019, (<https://2019.splashcon.org/track/splash-2019-SPLASH-E>) [Accessed 10/11/20]
- [5] SWEBOOK v3.0 Guide to the Software Engineering Body of Knowledge (<https://www.computer.org/education/bodies-of-knowledge/software-engineering>) [Accessed 8/11/20]
- [6] Dynamic Visualization and Time Markku Reunanen
- [7] Front-End Visualization Libraries: Recharts (<https://recharts.org/>) [Accessed 16/11/20]
- [8] Front-End Technology React documentation (<https://reactjs.org/docs/getting-started.html>) [Accessed 16/11/20]
- [9] Material-UI documentation (<https://material-ui.com/>) [Accessed 16/11/20]
- [10] Elasticsearch (<https://www.elastic.co/>) [Accessed 16/11/20]
- [11] "RESTful Web Services", <https://phppot.com/php/php-restful-web-service/> [Accessed 26/8/20]
- [12] "Nielsen's 10 Usability Heuristics", <https://uxdesign.cc/10-usability-heuristics-every-designer-should-know-129b9779ac53> [Accessed 24/10/20]
- [13] Elasticsearch to React Connectivity (<https://app.getpocket.com/read/2866109937>)
- [14] The Pragmatic Programmer 1st Edition, by Andy Hunt and Dave Thomas, Year 1999 by [Addison Wesley](#) [Accessed 18/11/20]