TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jaroslavna Khorosheva 222738IAIB

Development of a Web-based User Interface for Visualisation and Analytics of the Smart Open Fish Counter Database

Bachelor's Thesis

Supervisor: Jeffrey Andrew Tuhtan
Dr.-Eng.

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jaroslavna Khorosheva 222738IAIB

Veebipõhise kasutajaliidese arendamine Smart Open Fish Counter andmebaasi visualiseerimiseks ja analüüsiks

Bakalaureusetöö

Juhendaja: Jeffrey Andrew Tuhtan
Dr.-Eng.

Tallinn 2025

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Jaroslavna Khorosheva

04.06.2025

Abstract

The aim of this thesis is to create a full web application for visualising and analysing data from the Smart Open Fish Counter (SOFC) database. The application was developed for Tallinn University of Technology and is intended to help monitor fish populations, track migration patterns, and assess ecosystem health using data collected from underwater camera systems.

The application consists of a backend and a frontend developed from scratch. The backend was built using Java and Spring Boot, and the frontend was developed using Angular, providing interactive dashboards for detailed and aggregated fish data. The application also includes features such as a secure login system and automatically generated reports.

In the analysis section, the thesis describes the ways how the available data was picked and what filtering methods were used to support meaningful visualisation. Technological choices made before and during the development are described separately in the technology and development sections.

The result of the thesis is a functional full-stack web application that supports ecological monitoring through an intuitive and visually user-friendly interface.

The thesis is in English and contains 30 pages of text, 8 chapters, 6 figures, 4 tables.

Annotatsioon

Veebipõhise kasutajaliidese arendamine Smart Open Fish Counter andmebaasi visualiseerimiseks ja analüüsiks

Käesoleva bakalaureusetöö eesmärk on luua terviklik veebirakendus Smart Open Fish Counter (SOFC) andmebaasi andmete visualiseerimiseks ja analüüsimiseks. Rakendus töötati välja Tallinna Tehnikaülikooli jaoks ning selle eesmärk on aidata jälgida kalapopulatsioone, jälgida rändemustreid ja hinnata ökosüsteemi tervislikku seisundit veealuste kaamerasüsteemide abil kogutud andmete põhjal.

Rakendus koosneb nullist arendatud taustateenusest (backend) ja kasutajaliidesest (frontend). Taustateenus loodi Java ja Spring Booti abil ning kasutajaliides Angulari raamistikus, pakkudes interaktiivseid töölaudu detailsete ja koondatud kalandusandmete visualiseerimiseks. Rakenduses on lisaks funktsioonid nagu turvaline sisselogimissüsteem ja automaatselt genereeritavad aruanded.

Analüüsi osas kirjeldatakse, kuidas olemasolevaid andmeid analüüsiti ja milliseid filtreerimismeetodeid kasutati tähendusliku visualiseerimise saavutamiseks. Tehnoloogilisi valikuid, mis tehti enne arendust ja selle käigus, käsitletakse eraldi tehnoloogia- ja arenduspeatükkides.

Lõputöö tulemuseks on funktsionaalne täismahus veebirakendus, mis toetab ökoloogilist seiret intuitiivse ja visuaalselt kasutajasõbraliku liidese kaudu.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 30 leheküljel, 8 peatükki, 6 joonist, 4 tabelit.

List of abbreviations and terms

AI	Artificial Intelligence
API	Application Programming Interface
BfG	Bundesanstalt für Gewässerkunde, The German Federal Institute of Hydrology
CI/CD	Continuous Integration/Continuous Deployment
CORS	Cross-Origin Resource Sharing
CRUD	Create, read, update and delete operations
CSS	Cascading Style Sheets
DI	Dependency Injection
DTO	Data Transfer Object
E2E	End-to-End (testing)
GraphQL	Graph Query Language
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
MPA	Multi Page Application
PDF	Portable Document Format
REST	Representational State Transfer
SOFC	Smart Open Fish Counter
SPA	Single Page Application
SQL	Structured Query Language
TalTech	Tallinn University of Technology
TS	Typescript
UI	User Interface
UX	User Experience

Table of contents

List of figures

List of tables

1 Introduction

Modern environmental research increasingly relies on a vast amount of datasets collected through autonomous monitoring systems. In the case of aquatic ecosystems, tools like underwater cameras are used to record continuous videos of rivers and streams, creating detailed and often comprehensive datasets. This kind of big data is essential for understanding fish behaviour, monitoring populations, and identifying ecological patterns, but it only becomes valuable when it can be effectively used and analysed.

The problem of raw data itself is that it requires further post-processing to assess the statistical properties of the underlying data. Without the proper tools to explore and understand it, researchers may struggle to extract meaningful insights. One solution is to use well-designed software that can visualise data, creating a full view of the collected data and, therefore, turning it into something useful by providing key statistical parameters for further end-user assessment.

The main objective of this work is to provide a prototype web application to view statistical features, or analytics, of the Smart Fish Counter database. The existing database contains more than 400,000 videos and contains more than 30 European fish species. The information contained in the database are used by regulatory agencies, companies and researchers to evaluate fish biodiversity in rivers, which is required by the European Water Framework Directive (WFD), Habitats Directive (HD) and Eel Regulation (ER).

Current methods to evaluate this information use human raters, which means that processing the results of the database takes several months for each underwater camera location. The TalTech Smart Fish Counter technology suite encompasses underwater cameras (HydroCam [1]) and, since 2021, an automated computer vision solution for detecting fish and classifying species, size and migration behaviours. The prototype analytics web app developed in this work is an important first step for end-users, including the TalTech licensee I AM HYDRO GmbH, to be able to visualise key results such as the statistical distributions of fish size

and species over the course of the year.

To better set a clear direction for the project design, the author reviewed several existing systems that offer similar features in aquatic monitoring and data visualisation. These are:

1. Innovasea, Aquaculture Intelligence. They work with high-tech solutions for fish tracking in aquatic environments, and their dashboards focus on real-time monitoring, predictive insight, and data-driven farm management. [2]
2. RiverWatcher Daily. Their team provides fish monitoring dashboards for various rivers. The platform focuses on daily fish counts. [3]
3. FBIS, Freshwater Biodiversity Information System. FBIS aggregates freshwater biodiversity data from South Africa's rivers. While not directly focused on fish monitoring, they track river health, invasive species, and biodiversity trends. [4]

These tools are not direct competitors in a commercial sense, but they provide a good reference for visualisation, feature design, and user flow. Their approaches provided a big inspiration for shaping the scope and usability of this project.

2 Theoretical background

This chapter provides the theoretical foundation for the project, covering the three main topics: big data theory, data visualisation principles, and modern web application architecture. Each area was studied to create essential concepts and approaches which will lead to the development of a high-quality web-based dashboard for ecological monitoring. By understanding how large-scale data is handled, visualised, and delivered through the web, the project can better align with the I AM HYDRO team's ecological goals.

2.1 Big data characteristics and SOFC

Big data refers to extremely large, fast-growing, and complex datasets that traditional tools can no longer handle efficiently [5]. In the case of I AM HYDRO, the team applies big data solutions to fish monitoring using HydroCams [1] to achieve new ecological insights. These are underwater cameras which are equipped with artificial intelligence to automatically label fish species, behaviours, and environmental conditions.

2.1.1 The 6 Vs model

The 6 Vs model [6] is a widely used framework for understanding the main characteristics of big data, as well as it helps to understand that not all data is equally useful. By looking at the SOFC system, each of these six dimensions can be found, influencing how data is collected, managed, and interpreted.

- **Volume.** With a large amount of the collected data, we need to process and analyse more, which can be a challenge. For SOFC, millions of video frames and thousands of detection records across varying durations are collected in each region which all need to be processed.
- **Velocity.** It refers to the speed at which the data is generated and needs to be processed. HydroCams stream new video frames every second, creating a continuous flow of information. To deliver timely ecological insights, the system must be capable

of processing and analysing this data quickly.

- **Variety.** SOFC data includes both structured and unstructured types. Structured data, such as detection logs or time-stamped metadata, fits very well into tables and databases. In contrast, unstructured data includes video footage from HydroCams and manually written annotations that don't follow a consistent schema
- **Veracity.** Not all data is perfect. For the fish monitoring, factors like water turbidity can degrade video quality, and even AI models can mislabel fish species or behaviours. These uncertainties mean we need to treat data carefully and sometimes review data ourselves.
- **Value.** Data has value in business, but it's of no use until that value is discovered. In the case of SOFC, we could track fish populations, asses river health, and study different patterns in the aquantic environments. However, it is important to remember that not all data is useful and can lead to valuable information.
- **Variability.** Data needs to be up-to-date, as it reflects changing patterns over time. In the case of SOFC, fish activity isn't static, it varies with seasonal migration, river flow rates, and environmental conditions. For example, some species appear only at certain times of the year, or river turbidity may spike after heavy rain. To remain accurate and relevant, the system must account for these fluctuations and adapt its processing and visualisation accordingly.

This 6 Vs model demonstrates why the SOFC system qualifies as a big data application and why it needs modern tools to manage its complexity. Understanding these characteristics helps to analyse the collected data and turn this information into real business value: support data-driven decisions and improve monitoring.

2.2 Data visualisation principles

Having raw data is not enough to showcase the needed insights. First, a person must pick what to highlight, decide how to present it, and choose which visual elements best support the story the data is telling. This section describes the main principles that shaped the design for SOFC analytics dashboards and reporting tools, making sure users can quickly understand patterns and what kind of data is displayed.

2.2.1 Visual data processing

The human brain can process visuals between 6x and 600x faster than text. That is why clear and intuitive design is a priority when presenting data. Visual elements like colour, size, and position allow users to spot and understand a pattern instantly without conscious effort. [7] For instance:

- **Colour.** Using a specific colour for every criteria.
- **Size.** Larger elements draw the attention immediately. In a bar chart, a bigger chart emphasises the bigger count.
- **Position.** Placing the legend separately above the chart aligns with the common reading pattern and ensures users will read it first.

2.2.2 Layout and interactive design

A well-designed dashboard is not just about pretty charts, it is also about guiding users through data. The SOFC dashboard adheres to Schneiderman's Mantra: "Overview first, zoom and filter next, then details on demand" [8]. This approach means users first see a broad summary of activity and can zoom into specific sites, periods, or species. Interactive elements like tooltips also play a significant role in providing additional context upon hovering a selected datapoint, offering detailed information without cluttering the screen.

2.2.3 User experience heuristics

Another thing to take into consideration is good user experience (also known as UX). It reduces confusion and builds trust. Applying Jakob Nielsen's 10 usability heuristics [9] can significantly improve any dashboard's usability. While not all heuristics may be implemented in every design, incorporating those most relevant to the user's needs can lead to a more intuitive and effective interface. For instance:

- **Visibility of system status.** Loading spinners and subtle animations show users that data is still being processed, improving trust and clarity.
- **Match between the system and the real world.** The usage of familiar terminology and icons related to real-world concepts.
- **User control and freedom.** Users can easily undo actions or navigate back.

- **Consistency and standards.** Follow consistent design patterns for the components which behave similarly.
- **Recognition rather than recall.** Selection options are visible, which avoids relying on users' memory.

2.2.4 Visual Cognition Limits

Human cognitive capacity is limited, with individuals typically able to process 7±2 items simultaneously. For example, dashboards avoid overcrowding by displaying only essential widgets and allowing users to access additional details through interactive elements such as tooltips. [10] The responsive design can also go to this category, as screen space is restricted on tablets and smartphones. By adjusting the layout and scaling elements, it ensures that the dashboards remain readable and functional, and that all necessary items are displayed in the view.

2.3 Web-application architectures

In today's digital landscape, building and delivering software on the web has evolved from simple, page-oriented sites to big-scale platforms united with dozens of services, frameworks and data stores. Web application architecture refers to the structural framework that explains how all the various components, such as user interfaces, APIs, business logic, and infrastructure, fit together so your code can handle real-world demands. Combining all these parts into a cohesive system is crucial and often hard to achieve, but the outcome will be a robust application that supports scalability and long-term maintainability.

2.3.1 API paradigms: REST vs GraphQL

REST (Representational State Transfer) is a widely used architectural style that uses standard HTTP methods for communication between backend and client-facing applications. However, working with big data, REST may cause over-fetching or under-fetching, since each endpoint delivers a fixed data structure, which might include unnecessary fields or omit the important ones, requiring additional calls.

GraphQL, on the other hand, allows clients to specify exactly what data fields they need, reducing the amount of transferred data and improving performance. This selective

querying is particularly beneficial when dealing with complex and large datasets, where it helps optimise network usage and speed up data retrieval. [11]

2.3.2 Client-side patterns: SPA vs MPA

Single page applications (SPA) load a single HTML page and dynamically update content based on user interaction within the app. This approach produces a smooth experience after the first load and cut down the backend load after the shell is cached. However, SPAs can pose challenges for search engine optimisation (SEO), as content is rendered client-side and may not be easily indexed without additional tooling like server-side rendering or prerendering.

Multi-Page Applications (MPAs) reload the entire page on every navigation, which delivers a fast initial paint and excellent crawlability but introduces visible refresh delays and extra server round-trips. [12]

For internal tools like the SOFC dashboard, SEO is not a priority, making SPAs a suitable and efficient choice.

3 Stakeholders requirements

The first step in the project was to gather a clear understanding of the application's goals and constraints. To achieve this, meetings with the project stakeholders were held. These semi-structured interviews gave the team's expectations, usage goals, and technical constraints.

3.1 Stakeholder interview

All the project requirements were gathered through two interviews: the first conducted in November and the second in February, each lasting approximately one hour. Those involved were two people from TalTech and the I AM HYDRO team, who serve as data owners and future users of the system. Both sessions took place over a video call.

3.1.1 Key questions asked

Before the interview, the author prepared a set of questions to ensure the discussion would cover the most relevant functional aspects. It had a wide range of questions, but the following Table 1. represent the main ones used to understand stakeholders' needs and expectations.

Table 1. Main questions and their purpose.

Question	**Purpose**	**Answer**
Who are the primary users of the SOFC data?	To identify users' role and their usage	TalTech and I AM HYDRO end-users
What is the main purpose of the data visualisation?	To understand the impact of visualised insights	Assess the statistical properties
What specific metrics are the most important?	To prioritise what should be surfaced on the dashboard	Fish aggregation count and its parameters
What security measurements are needed to be applied?	To define the authentication need	Secure login system

Continues...

Table 1 – *Continues...*

Question	**Purpose**	**Answer**
Is offline usage required?	To assess technical limitations and define the need for reporting	Seen as useful but not essential

The answers to these questions helped to define what the system needed to do and shaped the functional and non-functional requirements presented in Chapters 3.2 and 3.3.

3.1.2 Interview outcomes

After the interviews, the author had a clear understanding of the project's purpose and the user's expected result. The key points are summarised below:

- Dashboard must visualise fish aggregation by region and support species-specific filtering for the ecological insight.
- SOFC data is private, so a secure authentication is required for all users.
- Full offline access was decided to be unnecessary.

These three elements shaped the initial prototype for the system.

3.2 Functional requirements

After the interviews were conducted, the author set the project's primary features. Each will guide and serve as a reference point during the development and testing.

- The dashboard should include a regional aggregation to visualise fish activity across monitored rivers.
- The dashboard shall provide species-specific charts, displaying metrics, trends and detection data for the selected fish types.
- The system must implement a secure login using JWT (JSON Web Tokens) and a refresh token for session continuity.
- The system shall include an automatic PDF report generator that will allow users to export charts and summaries as an alternative to offline access.

Together, those components address all the key requirements gathered from the stakeholders

and the prototype, and form the foundation for the system's design.

3.3 Non-functional requirements

Beyond functionality, the application must also meet a set of non-functional elements that define its quality: performance, scalability, maintainability, and usability. These can be detailedly seen in Table 2.

Table 2. Non-functional requirements.

Attribute	**Target**
Performance	Dashboard view should load in under 2 seconds (median)
Scalability	The system should support data growth without rewriting the code
Maintainability	The codebase must follow strict linting and style rules. A developer wiki and clear documentation should enable a new contributor to become productive within one week.
Usability	The application must follow Jakob Nielsen's 10 usability heuristics and maintain a responsive design for screens down to 1024px.

Special attention is given to maintainability. The author views long-term code quality as a key project success factor, especially given potential team changes and handovers. Two dedicated chapters (see: Chapter 4.2.5 and Chapter 4.3.1) expand on this topic, covering how these sets of rules affected the development workflow in detail.

4 Technology overview and implementation

This chapter explains what technologies were chosen before and during the development of each layer of the system's architecture, and how each technology was applied within the project. These choices were picked by the key requirements mentioned in Chapter 3. Later chapters will provide a more detailed breakdown of their implementation and integration throughout the system.

4.1 Data layer

PostgreSQL was already in use at I AM HYDRO for the Smart Open Fish Counter (SOFC) database, so the project continued using it unchanged.

4.2 Backend stack

Java 23 was a primary choice when selecting a backend technology due to its portability, scalability, and security features. It has cross-platform support and a mature ecosystem, giving access to various libraries, frameworks, and tools that can speed up the development process. In addition, its object-oriented design allows developers to write reusable code, making the system more modular and easier to maintain. [13]

Furthermore, Java is great at managing large datasets and supporting high traffic in backend systems. It also integrates well with relational databases like PostgreSQL, making it a solid choice for the SOFC analytics platform.

4.2.1 Framework

To make development speedier, it was decided to implement Spring Boot 3, a framework that simplified Java backend development by offering pre-configured defaults and starter modules [14]. Spring Boot also has a seamless integration with PostgreSQL and compatibility with tools like Docker, which further enchanes its suitability for the project.

4.2.2 Security

For the security layer, Spring Security was integrated. It goes along with Spring Boot, providing authentication and authorisation mechanics, and ensuring that the data transfers between the backend and other client-facing applications are secured.

JWT (JSON Web Token) authentication was implemented with refresh tokens to manage user login sessions effectively. This approach was chosen as it gives both security and improves UX by minimising the need for frequent logins and checking that expired tokens are not misused. Upon successful login, the system issues an access token with a short lifespan and a refresh token with a longer validity period. When the access token expires, the refresh token goes into the implementation to obtain a new access token without requiring the user to re-authenticate. [15]

4.2.3 API styles

A hybrid API strategy was adopted as it delivers the two main usage patterns described in Chapter 2.3.1. REST endpoints handle CRUD (create, read, update and delete) operations because they follow a simple structure, aligning with HTTP methods and therefore making them ideal for standard data manipulation tasks.

For the analytics dashboad, however, GraphQL works better, as its client-driven queries allow the client to request exactly the fields it needs in one round-trip. This keeps network traffic low and speeds up data load times for complex charts.

4.2.4 Reporting

To create PDF generation, the project pairs Thymeleaf with OpenHTMLtoPDF. Thymeleaf was chosen for its seamless integration with Spring Boot and its ability to render HTML templates with almost no extra configurations [16]. Once the server renders a template, OpenHTMLtoPDF converts the rendered HTML chart and chosen data into PDFs using CSS styling. This combination allows for the easy management of templates and generating documents using SOFC data and visualised charts.

4.2.5 Documentation

To enhance API usability and maintainability, the author integrated OpenAPI into the project. This generates detailed, interactive API documentation directly from the specification and helps developers explore endpoints, view API examples, and understand API behaviour without needing additional resources [17].

By adding dynamic documentation, it provides that as the project evolves in the future, future developers can easily understand and interact with the API. This will significantly reduce the time to analyse the codebase and help new team members get up to speed faster.

4.3 Frontend stack

Angular 18 was selected over alternatives like React due to its built-in structure and opinionated architecture, which promotes consistency across the codebase. This framework enforces a modular approach, making it easier for different developers to follow a unified coding style. Additionally, it has a strong dependency injection system, which encourages clean, non-duplicated code. Its tight integration with TypeScript (TS) provides static typing, which helps catching errors early in the development process. [18]

4.3.1 Code quality and formatting

To maintain a clean and consistent codebase, Prettier and ESLint were also implemented. Those tools automatically format code and enforce linting rules to prevent style drift. This was especially important for the project handover, ensuring that future developers can easily work with the codebase, following the same style standards.

4.3.2 Dashboard library

Apache ECharts was integrated into the frontend for interactive data visualisations, as it has a native Angular wrapper, ngx-echarts, which allows seamless integration with the existing Angular 18 stack. Echart also offers a variety of chart types, making it well-suited for the SOFC dashboard. It also offers strong performance and a flexible library for handling large datasets, making it an ideal choice for this project.

4.3.3 PDF export

To meet the requirement for exporting dashboard snapshots, the project included a client-side solution using FileSaver.js. This enables the frontend side to capture the current state of the dashboard and save it as a PDF file directly from the browser.

4.3.4 Styling and interaction libraries

For styling and UI enhancements, the following libraries were selected for the main components and interactive aspects of the application:

Table 3. Used libraries.

Category	**Libraries**	**Why chosen**
Components	Bootstrap 5, DaisyUI	Rapid layout prototyping and responsive design capabilities.
Icons	Font Awesome	Broad icon set with tree-shakable imports to minimise bundle size.
Animation/Motion	Lenis (smooth scroll), Lottie Web + ngx-lottie	Improves UX with lightweight, designer-friendly JSON animations.
3-D Visualisation	three.js + @types/three	Opens possibilities for future enhancements like interactive underwater scene rendering.

4.3.5 Authentication helpers

To support secure communication with the backend, an HTTP interceptor paired with ngx-cookie-service was implemented to automatically attach JWTs to outgoing requests. This also keeps the token in storage and refresh logic, keeping the frontend in sync with the backend authentication.

5 Development

This chapter describes the development process, including technical preparation, data handling, and implementation decisions that shaped the application architecture.

5.1 Data preparation

Before development could begin, it was necessary to understand the database structure and contents of the SOFC database. The database is extensive, containing many tables and tens of thousands of rows, including over 25,000 entries specifically related to fish monitoring.

The first step was to identify and choose only the tables relevant for the analytic dashboards. Once those were selected, further filtering had to be done, which was eliminating columns whose fields were not useful for visual analysis. Such data might have potential for future development, but in the current project, it stored no value. For example, metadata such as "last modified" timestamps, while valuable for data tracking but offers no insight into fish behaviour or river conditions.

During this filtering process, it was also observed some data consistency issues. Few fields did not follow a strict schema, particularly in older records. For instance, instead of a level indicator "LOW", which was used in river monitoring, a few cases had it under "Low". Although this type of inconsistency may seem minor, it can lead to errors during aggregation or visualisation, especially in charts where category names must match exactly.

Since the issues could be accidentally repeated, these problems had to be addressed at the backend service level. The best way was to implement filters and data transformation logic that classify and normalise values into predefined types. This ensures that the data is clean and standardised before it reaches the frontend and reporting layer, reducing the risk of inconsistency in visualisations.

Due to the fact that the Smart Open Fish Counter database is private, the exact structure,

including names, column names, and schema details, cannot be disclosed in this document. However, to provide context, the diagram below presents a simplified view of the fish-related data model that this project primarily worked with.

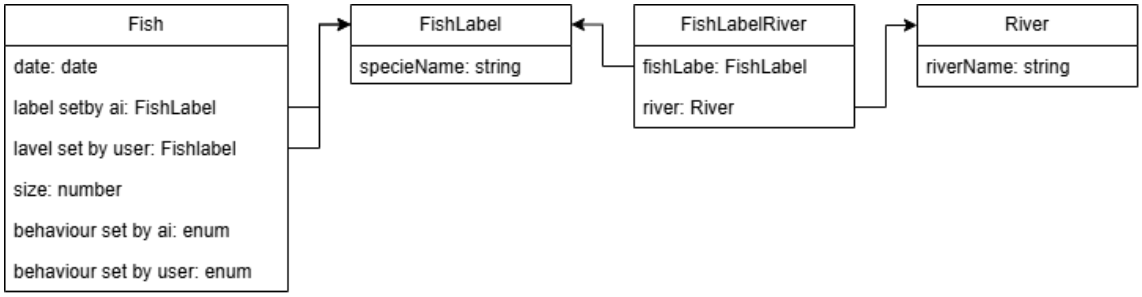


Figure 1. Fish-related data model and relationships.

It is important to note that this is not the actual database schema, but rather an abstract representation to illustrate the core data relationships relevant to the dashboard.

5.2 Backend architecture

The first step in setting up the backend was to initialise the project by adding all required dependencies. This included Spring Boot starters, PostgreSQL driver, JWT authentication, GraphQL, and OpenAPI for documentation. Once the project structure was in place, the development moved to creating the core components on which the system would be working.

The progress began with defining entity classes, which in Spring represent the structure of database tables. Each entity was created for one of the selected tables and only had fields that were relevant for the dashboard. For the inconsistent or non-standardised data (e.g., "Low" instead of "LOW"), the author created Java enum classes to strictly define expected values. This approach ensures the detection of schema violations at runtime and helps to fix them using custom converters (e.g., automatically convert "Low" to the standard "LOW") before processing data.

For each entity, a corresponding repository interface was created. In Spring, repositories are used to interact with the database, and by extending JpaRepository in the backend layer, the system gains built-in support for common queries like findAll(), findById(). This allows the author to use existing methods without writing their query logic.

To prevent sending raw entity objects (which represent the database) to the frontend, the author created DTOs (Data Transfer Objects) for each entity. This is important because exposing entity objects can:

- Reveal the internal database structure to the client.
- Unintentionally leak sensitive or unnecessary fields.
- Make the system more tightly coupled and harder to maintain.

To map correctly every entity between its DTO, MapStruct (a Java annotation processor that automatically generates mapping code at compile time), was used for its type-safe code generation. Additionally, Lombok (a library that auto-generates getters, setters, constructors, and builders) was also included to improve the clarity of the code and keep the codebase clean and readable.

The next layer added was the service layer, which is the main part in handling all business logic. In Spring, service classes are used to manage functionality, usually by using repositories and preparing data to be returned via DTOs. Additionally, the author implemented custom exceptions and a global error handler, which ensures that all errors follow a consistent format:

- Timestamp
- HTTP status
- Exception type
- Internal error log
- Developer-friendly message

This structure makes debugging and error tracking significantly easier and faster.

The final layer between an entity and a client-facing application is the controller layer. It is responsible for handling API requests.

Short explanation of how this architecture works:

1. The controller receives a request for fish data.
2. The service retrieves the relevant entity via a repository.
3. The entity is mapped to a DTO, which includes only necessary fields like name,

species, and river.

4. The DTO is returned to the frontend through the API, while the internal entity remains encapsulated and protected.

Here is a diagram of the backend architecture to visualise this flow.

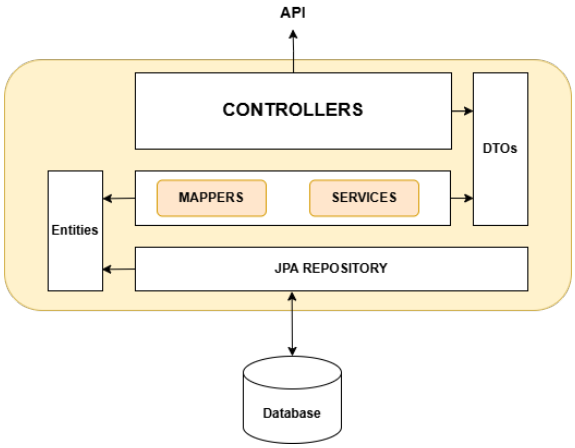


Figure 2. Backend architecture diagram.

As of now, the backend consists of 12 entity classes, 21 DTOs, 17 enums, 12 mappers, 8 repositories, 6 service classes, and 6 controllers.

5.3 Backend security implementation

The security layer is based on JWT authentication, which is described in Chapter 4.2.2. To implement JWT-based security, two key supporting classes were created: JwtRequestFilter and JwtTokenProvider.

The JwtTokenProvider class is responsible for generating and validating access and refresh tokens. It securely loads the signing keys from environment variables and decodes them using Base64. Tokens are generated using a builder pattern with custom claims, timestamps, and expiration logic.

The JwtRequestFilter is a Spring Security filter that intercepts each HTTP request to extract and validate the JWT token. It checks for a valid Authorisation header, extracts the token, and then verifies that the token is present, not expired, and the token type is correct.

Additionally, CORS (Cross-Origin Resource Sharing) is configured in the service layer through a SecurityConfiguration class. It allows defining which frontend domain can be

trusted to send requests to the backend, and also requires authorisation to other APIs.

5.4 Backend performance and optimisation

Having the volume and complexity of data, it was clear that the smart service for data transformation was a priority. To support this need, particularly for the fish-related API methods introduced in Chapter 4.2.3, the author decided to implement GraphQL for those endpoints. It ensured that the payload is small, improving response speed compared to REST endpoints.

One key implementation was the fishDailyAggregate query. This query returns time-series data on fish behaviour, aggregating values by date and classification (e.g., UP, DOWN, IN, etc.).

Although GraphQL offered many advantages, it also gave unexpected challenges. In early development, the author encountered frequent runtime errors due to null values. The issue was difficult to trace, especially when working with such an enormous database and was further complicated by the author's limited experience with GraphQL's strict type and field rules. As a result, it was decided to pick a new approach by using projections, which used the existing error handling and service classes.

A projection in Spring is an interface-based class, in the current case, allowing defining only the fields required for the specific response. By using them, the author bypassed full entity loading and retrieved only the relevant data directly from the repository, using a customised query.

This approach was especially effective for behavioural statistics, where counts of fish activity, labelled by both AI and user, were grouped by river. The logic for computing the counts was placed in the repository, using a native SQL query, making the database engine handle the heavy lifting and significantly boost performance. If a similar logic were applied to the server layer instead, its performance would be much slower, as it would need to calculate these counts manually by processing large datasets in memory, which could increase the risk of runtime errors and inefficiencies.

One of the projections, which was created, was RiverBehaviourAggregation. It had three

fields: river name, user count, and AI count. This interface was linked directly to the repository's custom native SQL query, which stored the query result and therefore was treated similarly to DTO. Although the service layer continued to orchestrate the logic and handle exceptions, it now operated on these lightweight projection objects. This architectural shift led to a more efficient, scalable, and maintainable solution for handling high-volume data.

To further optimise performance, the API endpoints were extended with different filter parameters, including river name, species, date range, and time interval. However, unlike behavioural counts, which are calculated directly within the query and returned as a result, returning detailed fish DTOs based on filters would still be hard to process all at once. To address this, the author implemented pagination, a technique that breaks large result sets into smaller, for fish-related data. This approach allowed the system to process only a subset of filtered results at a time, and when total counts were needed, the system aggregated the results page by page.

5.5 Backend generated reports

For the automatically generated reports, a new endpoint was added that accepts a ReportDTO from the frontend, containing metadata like the report title, notes, and a list of chart images encoded as data URLs. After receiving a request, the service layer gets delegated with a new task to render the data into an HTML template using Thymeleaf. Then this HTML is passed to OpenHTMLtoPDF, which converts it into a properly styled PDF and streams it directly to the client as a download using Spring's ResponseEntity.

This approach ensures consistent formatting, allows for easy customisation of the report layout, and supports offline access.

5.6 Frontend architecture

The frontend of the project was built using Angular version 18, and its setup began by installing all required libraries, mentioned in Chapter 4.3.4. Then the author set a clean and scalable folder structure by dividing the codebase into three main directories: core, shared, and views.

The core folder includes application logic and configurations, containing essential files that define business logic and type safety of the application:

- Enums. These are predefined, strongly typed constants used throughout the application. Some replicate enums from the backend (e.g., behaviour types), while others are frontend-specific, like label groupings or period filters.
- Guards. Guards control route access in Angular. This project has an AuthGuard, which restricts access to certain view pages unless the user is authenticated.
- Models. These are TypeScript interfaces that describe the shape of objects received from the backend (e.g., FishDTO, RiverStatsDTO). They serve a purpose to get defined objects that the backend sends and have strict type configuration for type safety when processing API responses..
- Services. Services manage all communication between the frontend and backend. Each service calls specific API endpoints and returns typed results (using the models above) to the components that need them.

The shared folder stores UI components and utilities that are reused across different parts of the application, such as common components like a loader, fish animations, or "not found" messages. These help maintain design consistency and avoid code duplication.

The views directory contains all the main page components. Each view corresponds to a route and can include nested components specific to that section. The structure is hierarchical (meaning if a fish monitoring component includes a chart component, that chart component will be located in the fish folder) to maintain modularity and clarity. The structure diagram can be seen in Figure 3.

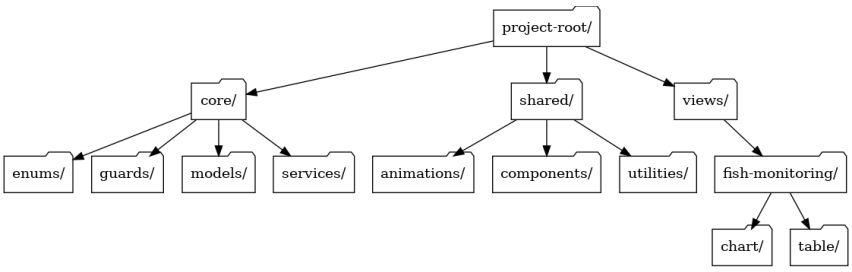


Figure 3. Frontend structure diagram.

This structured approach ensures the codebase is easy to navigate and maintain, especially if the project will be handed over.

Another important aspect that shaped the frontend architecture is the consistent use of Angular's dependency injection (DI) and component communication patterns. A significant portion of development time was spent designing a clean, maintainable structure for how components interact, particularly between parent and child components.

To avoid code duplication and promote reusability, the author relied heavily on @Input() and @Output() decorators. In Angular, @Input() is used to pass data from a parent component to a child, while @Output() allows the child to emit events or data back to the parent. This pattern enabled the creation of flexible, self-contained components that could adapt their content based on input parameters without duplicating logic. This was especially useful for components that visualise data from API endpoints. As mentioned above, as the project uses a hierarchical folder structure, all API calls are handled at the parent component level, and child components receive only the specific data they need.

Practical example: instead of creating two nearly identical components, one to show a bar chart of fish sizes and another to show species distribution, the author creates a single, reusable chart component. This component accepts a data input like this:

```
<app-bar-chart [data]="sizeData"></app-bar-chart>
<app-bar-chart [data]="speciesData"></app-bar-chart>
```

Figure 4. Reusable bar chart component usage with different data inputs.

Here, sizeData and speciesData are prepared in the parent component via service calls. The child chart component doesn't process the data, it just renders what it's given.

This pattern was a foundational architectural choice where the author devoted considerable effort to planning how services, parent views, and nested components would interact. By enforcing a strict data flow and structured component hierarchy, the application is easier to extend, simplifying future work for development.

5.7 User interface and responsiveness

For the user interface (UI), the design was built using the libraries introduced in Chapter 4.3.4. The first major decision was to define the overall layout of the application. After

evaluating different options, the author chose a navbar paired with a sidebar layout. This structure offers clear and intuitive navigation, while also being highly scalable, unlike a navbar-only design. The sidebar can accommodate a growing list of navigation items without overwhelming the user or cluttering the interface. A prototype of the default layout is shown in Figure 5.

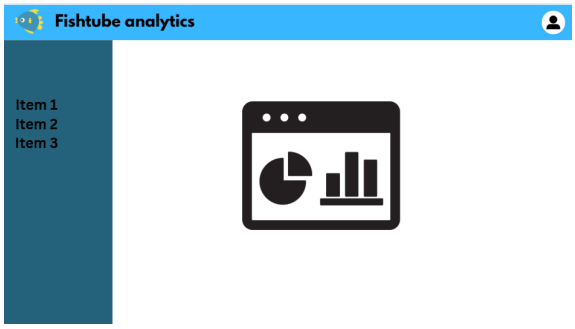


Figure 5. Prototype of the default layout.

Once the desktop layout was established, special attention was given to responsive design, ensuring the application would remain usable and visually clean on smaller screens. The initial idea was to minimise sidebar items to icons os smaller screens. However, this was rejected, as even a small sidebar would take up horizontal space, which would significantly reduce the visibility of dashboard charts.

As a final and more user-friendly solution, the sidebar was repositioned to the bottom of the screen. This design brings several advantages:

- It preserves vertical space for charts and dashboards.
- It offers quick, thumb-friendly access to navigation items without requiring large hand movements.

This bottom-oriented layout proved to be more intuitive and efficient for mobile users. A prototype of the responsive layout is provided in Figure 6.

Figure 6. Prototype of the mobile layout.

5.8 Dashboard visualisation and reporting

Using Angular service, the author developed a variety of chart components to visualise fish and environmental data. These components received their input data via parent-child injection, which was mentioned in Chapter 5.6. Visualisation was implemented using Apache ECharts, integrated through the Angular wrapper ngx-echarts, as described in Chapter 4.3.2.

In addition, each dashboard view was paired with filter components for parameters such as species name, time period, and date range. Furthermore, when a specific river page was opened, the view automatically filtered data by the selected river name, showing a full insight into only the selected site, reducing the need for duplicative selections.

The design of the dashboards was also influenced by the visual cognition principles discussed in Chapter 2.2.1. For instance, colour was used to distinguish different species, and larger chart elements were used to highlight important trends (e.g., bar size representing species count), helping users quickly detect meaningful patterns. In addition, chart legends were positioned consistently above the visual elements to align with the natural reading flow of the end-users.

User experience heuristics outlined in Chapter 2.2.3 also shaped the dashboard interface. Elements like loading spinners were used to reflect system status and show transparency during data loading. Selections for species and rivers were provided as predefined options, so users did not need to type manually, reducing input errors and improving usability. Additionally, the UI includes navigation shortcuts and back buttons, allowing users to

quickly move between components and views.

For reporting, a client-side export feature was implemented using FileSaver.js, as explained in Chapter 4.3.3. The frontend constructed a custom DTO containing the relevant data snapshot and sent it to the backend. From there, the backend handled PDF rendering, following the process detailed in Chapter 5.5, which included using Thymeleaf templates and OpenHTMLtoPDF to generate downloadable reports.

6 Evaluation and results

6.1 Performance tests

To evaluate performance improvements, the author selected one of the most active rivers in the dataset for testing (around 8,500 entries). All tests were conducted using the author's home Wi-Fi network, with an approximate speed of 191 Mbps. The response times's median for key API endpoints were compared before and after backend optimisations, such as pagination and projections.

Table 4. API response time.

API endpoint	**Without optimisation**	**With optimisation**
Get fish aggregate count for rivers by period	12 seconds	736 ms
Get fish data by species	12 seconds	79 ms
Get the fish behaviour count by river	Not tested	183 ms

6.2 Dashboard coverage

The system currently provides 22 charts, covering different perspectives of fish monitoring and environmental conditions:

- 11 charts related to river-wide monitoring, including:
  - Fish behaviour in the river
  - Species variety
  - Fish count by hour and by month
  - Injured fish statistics
  - Fish size distribution
- 7 charts focused on environmental conditions, such as turbidity, biofilm presence.
- 4 charts tailored to individual fish species, offering detailed insights about the species.

Example views of the dashboards are included in Appendix 3.

Additionally, the author also developed a custom chart prototype. Currently, it allows users to:

- Select their filters (e.g. fish species and river).
- Choose from three chart types: pie chart (behaviour), bar chart (size), and bar chart (activity).
- Save their filter configuration to localStorage.

Saving to localStorage has its limitations, so the author added functionality for exporting and uploading filter configurations as JSON files. This also leaves room for the future integration with Redis (an in-memory data structure store that can be used as a database). Although still in the prototype stage, this feature has received early interest from stakeholders. More discussions are required to refine its scope and align it with the team's needs.

6.3 Report generation

At present, the system supports two types of downloadable reports:

- River report. Covers full behavioural and environmental trends for the selected river, including high and low detection points.
- Species report. Gives activity patterns of one fish species, including high and low detection points.

Each report includes visual charts, summary statistics, and a PDF export as described in Chapter 5.5.

6.4 Usability feedback

A 4-hour demonstration session was conducted with two people from TalTech and one team member from I AM HYDRO. During the session, participants were able to explore dashboards freely and test the filters.

Overall, the feedback was positive:

- Users appreciated the visual clarity and the speed of filtering.
- They suggested additional features and filters, which would help them analyse data

more deeply.

These insights will guide the next round of improvements and UI adjustments.

6.5 Deployment outcome

To simplify the deployment process for I AM HYDRO, the author implemented automatic deployment using GitHub Actions and Docker. The workflow consists of four pipelines:

1. Build the Spring Boot backend
2. Build the Angular frontend
3. Package both as Docker containers
4. Push final images to a Docker Hub repository

This CI/CD setup ensures that every new code commit triggers a clean, consistent build and deployment. It reduces manual work, eliminates human error, and makes it easy to deploy updated versions of the system to production or test environments.

To further improve the development process, the author also explored the use of key vault solutions (key vaults store secrets/sensitive values outside the source code, avoiding potential leaks and improving security). The goal was to use a solution similar to Azure Key Vault, and during the research, tools Doppler and AWS Secrets Manager were considered. However, most production-ready offerings required a paid subscription or complex setup beyond the scope of this project.

As a result, no automated secret management is currently implemented, and all sensitive values must be configured manually through environmental variables during deployment.

7 Future work

While all required and non-required features were successfully implemented within the project goals, the author sees multiple opportunities to improve and extend the application system further. As the core architecture has been intentionally designed with scalability in mind, it allows new features and improvements.

One main focus for the future development is the custom dashboard feature, intoduced in prototype form during this project. Expanding this feature would involve allowing users to select additional filters, choose from a broader variety of chart types, and fully configure dashboard layouts, without adding new code. However, due to the large amount of data, implementing a fully stable and flexible version within the current scope was a problem. The existing prototype includes a few data-fetching bugs, which would need to be investigated and resolved in the future.

To ensure long-term maintainability and code quality, the author also recommends integrating SonarQube into the CI/CD pipeline. This would perform automated code analysis, highlighting potential bugs before new features are deployed.

Another important step is to add tests to both the backend and the frontend, ideally achieving at least 80 per cent unit and end-to-end test coverage. This would help detect errors early, especially if new features are added or existing logic is changed.

Finally, the current Docker-based deployment could be extended into a Kubernetes-based deployment pipeline. With this, the team could use an auto-deploy cluster, and secrets could be securely injected using a key vault, such as Azure Key Vault.

Together, these improvements would increase the project's and its production level quality for TalTech and end-users including I AM HYDRO.

8 Summary

The primary goal of this thesis was to create a full-stack web-based user interface that visualises and analyses data held in the Smart Open Fish Counter (SOFC) database. A secondary goal was to create the fundamentals of the project for future I AM HYDRO monitoring solutions by implementing clean architecture, automated deployment and comprehensive documentation.

The first goal was reached through a series of structured steps. The work began with analysing the raw SOFC data, at the same time identifying data-quality issues and setting up functional and non-functional requirements with stakeholders. Next, suitable technologies were selected with a strong analysis of every implementation: Spring Boot and PostgreSQL on the backend, Angular 18 and TypeScript on the frontend, and Apache ECharts for interactive dashboards. Security was implemented with JWT access and refresh tokens, protected by an Angular HTTP-interceptor and route guards. Performance was implemented through projections and paginated endpoints.

The secondary goal was also achieved. GitHub Actions pipelines build, containerise and publish both the backend and the frontend services to Docker Hub, giving an easy deployment. A detailed wiki and strict code-style rules also ensure that new developers can onboard quickly and extend system without duplication.

Users were able to test the analytics web app and confirm that core functionality was successfully achieved within the BSc project timeline. In summary, the completed solution meets the stakeholders' requirements and has undergone preliminary testing by one end-user.

References

[1] I AM HYDRO. *HydroCam*. [Accessed: 24.02.2025]. URL: https://iamhydro.com/en/fishmonitoring/hydrocam/.

[2] *Innovasea, Aquaculture Intelligence*. [Accessed: 01.04.2025]. URL: https://www.innovasea.com/aquaculture-intelligence/biomass-estimation/.

[3] *RiverWatcher Daily*. [Accessed: 01.04.2025]. URL: https://www.riverwatcherdaily.is/Rivers.

[4] *FBIS, Freshwater Biodiversity Information System*. [Accessed: 01.04.2025]. URL: https://jrsbiodiversity.org/welcome-to-fbis/.

[5] Michael Chen. *What Is Big Data*. [Accessed: 24.04.2025]. URL: https://www.oracle.com/big-data/what-is-big-data/.

[6] Gouranga Jha. *Understanding the 6 Vs of Big Data: Unraveling the Core Characteristics of Data-Driven Success*. [Accessed: 24.04.2025]. URL: https://medium.com/@post.gourang/understanding-the-6-vs-of-big-data-unraveling-the-core-characteristics-of-data-driven-success-38e883e4c36b.

[7] Data Scientist Sahin Ahmed. *Essential Principles for Effective Data Visualization*. [Accessed: 24.04.2025]. URL: https://medium.com/thedeephub/essential-principles-for-effective-data-visualization-a13f05d22c39.

[8] Brooke Fitzgerald. *Schneiderman's Mantra*. [Accessed: 24.04.2025]. URL: https://hampdatavisualization.wordpress.com/2016/02/26/schneidermans-mantra/.

[9] Meghna Basak. *Simplified: Jakob Nielsen's 10 usability heuristics*. [Accessed: 24.04.2025]. URL: https://medium.com/thedeephub/essential-principles-for-effective-data-visualization-a13f05d22c39.

[10] Hardik Dewra. *The 7+2 Rule: The REAL Science Behind Miller's Law that will shock you your deepest memory!* [Accessed: 24.04.2025]. URL: https://medium.com/design-bootcamp/the-7-2-rule-the-real-science-behind-millers-law-that-will-shock-you-your-deepest-memory-4a35be25bb3b.

[11] *GraphQL is the better REST*. [Accessed: 26.04.2025]. URL: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

[12] Tim Davidson. *Single Page Application (SPA) vs Multi Page Application (MPA): Which Is The Best?* [Accessed: 26.04.2025]. URL: https://cleancommit.io/blog/spa-vs-mpa-which-is-the-king/.

[13] Vinayak Bhosale. *8 Reasons to Use Java for Backend Development*. [Accessed: 12.02.2025]. URL: https://www.thinkitive.com/blog/java-for-backend-development/.

[14] Pujan Zaverchand. *10 Reasons Why You Should Use Spring Boot*. [Accessed: 12.02.2025]. URL: https://www.sigmasolve.com/blog/10-reasons-why-you-should-use-spring-boot/.

[15] GeeksForGeeks. *JWT Authentication With Refresh Tokens*. [Accessed: 12.05.2025]. URL: https://www.geeksforgeeks.org/jwt-authentication-with-refresh-tokens/.

[16] Kelechi Divine. *Thymeleaf and Spring-boot. A short story.* [Accessed: 10.05.2025]. URL: https://okoroaforkelechi.medium.com/thymeleaf-and-spring-boot-a-short-story-ebafddb0b443.

[17] Rupesh Garg. *OpenAPI Specification vs Swagger: Understanding the Key Differences*. [Accessed: 10.05.2025]. URL: https://www.frugaltesting.com/blog/openapi-specification-vs-swagger-understanding-the-key-differences.

[18] *Angular vs. React: Which Should You Choose for Frontend Tech?* [Accessed: 10.05.2025]. URL: https://fullscale.io/blog/angular-vs-react/.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Jaroslavna Khorosheva

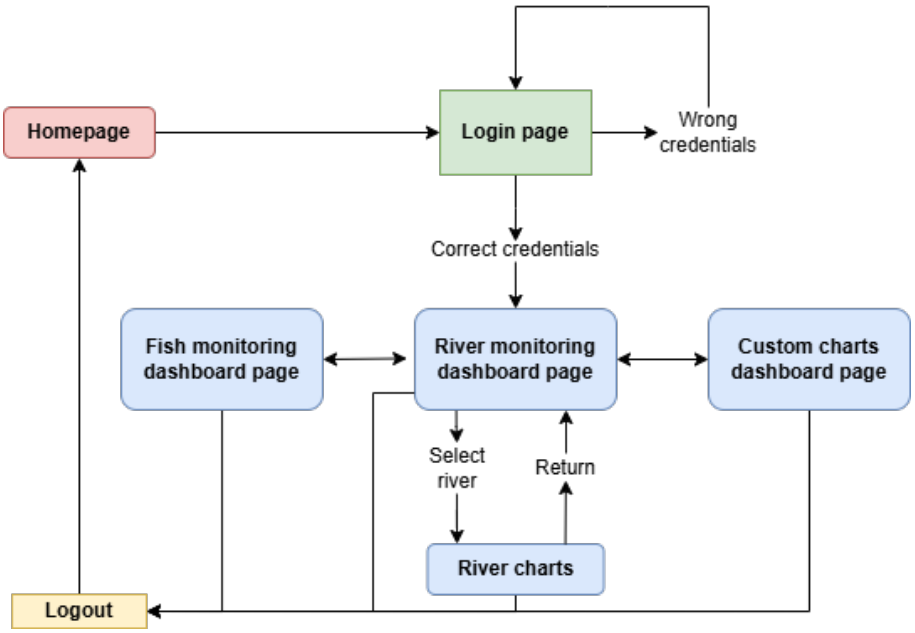
1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Development of a Web-based User Interface for Visualisation and Analytics of the Smart Open Fish Counter Database", supervised by Jeffrey Andrew Tuhtan
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

04.06.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – Frontend routing structure

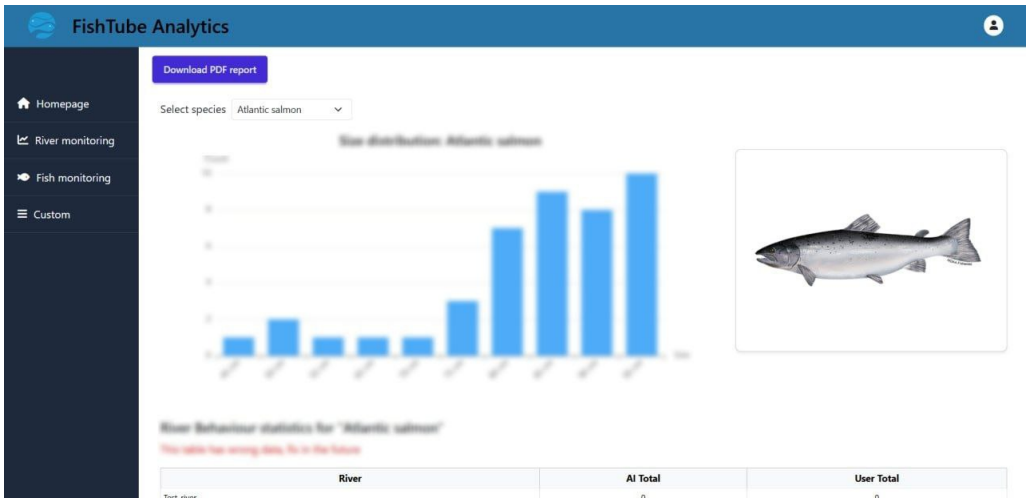
This appendix presents the full routing structure of the frontend application, showing all available paths and views defined within the Angular framework.



Appendix 3 – Dashboard interface

This appendix presents sample views of the river monitoring and fish monitoring dashboards. All data has been blurred to protect private information.

Selected river dashboard view.



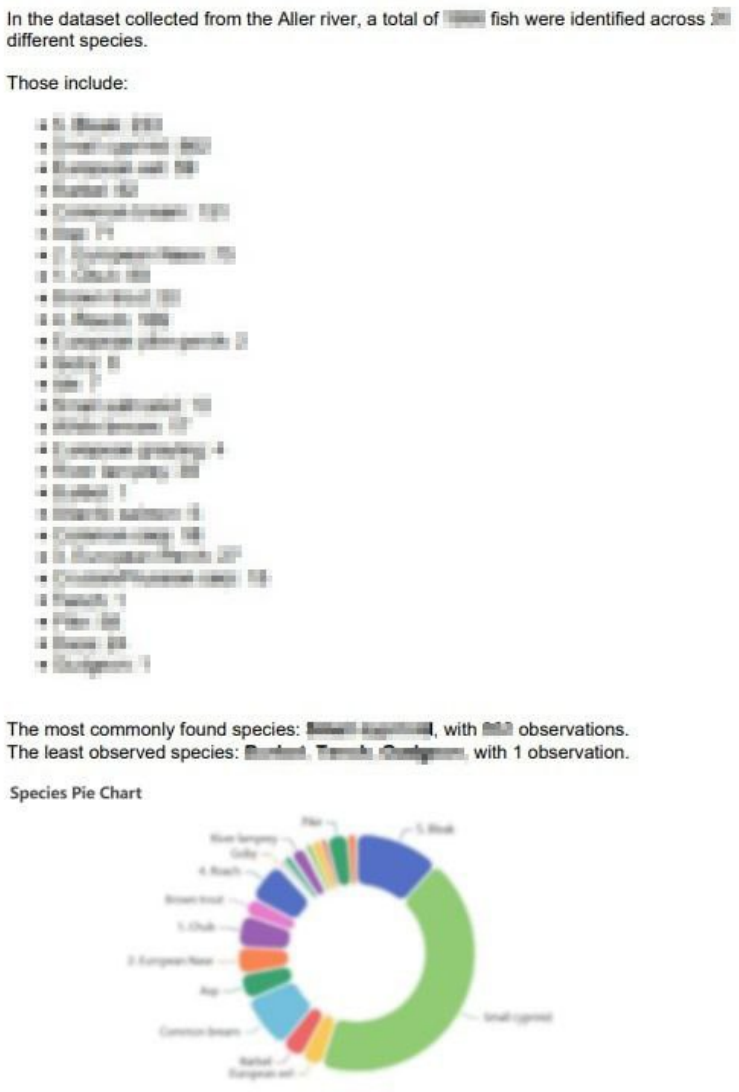
Selected fish statistics view.

Selected river dashboard mobile view.

Selected fish statistics mobile view.

Appendix 4 – Automatically generated report example

This appendix presents sample pages from the river monitoring and fish monitoring reports. All data has been blurred to protect private information.



Selected river report view.

I AM HYDRO GmbH

Fish report for Atlantic salmon

Generated automatically on 5/13/2025.

This report presents analytical insights into the fish species **Atlantic salmon**, based on data collected from monitored river environments.

It includes a distribution of observed fish sizes and behavioural activity across different rivers. These visualizations aim to highlight the prevalence, diversity, and activity patterns of the selected species as captured by the system.

All data is based on real-time monitoring and species labelling from both user and AI perspectives.

Reference image for the species **Atlantic salmon**.

This summary lists the rivers where **Atlantic salmon** was observed, based on labelled behaviour entries. Each fish is counted once per source — either from AI detection or user labelling.

Selected fish statistics report.