Faculdade de Ciências da Universidade de Lisboa

Projeto PSI
Software Architecture Document (SAD)

CONTENT OWNER: Group 7

DOCUMENT NUMBER:	RELEASE/REVISION:	RELEASE/REVISION DATE:
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•

Table of Contents

1 Documentation Roadmap

The Documentation Roadmap should be the first place a new reader of the SAD begins. But for new and returning readers, it is intended to describe how the SAD is organized so that a reader with specific interests who does not wish to read the SAD cover-to-cover can find desired information quickly and directly.

Sub-sections of Section 1 include the following.

- Section 1.1 (“Purpose and Scope of the SAD”) explains the purpose and scope of the SAD, and indicates what information is and is not included. This tells you if the information you’re seeking is likely to be in this document.

- Section 1.2 (“How the SAD Is Organized”) explains the information that is found in each section of the SAD. This tells you what section(s) in this SAD are most likely to contain the information you seek.

- Section 1.3 (“Stakeholder Representation”) explains the stakeholders for which the SAD has been particularly aimed. This tells you how you might use the SAD to do your job.

- Section 1.4 (“Viewpoint Definitions”) explains the *viewpoints* (as defined by IEEE Standard 1471-2000) used in this SAD. For each viewpoint defined in Section 1.4, there is a corresponding view defined in Section 3 (“Views”). This tells you how the architectural information has been partitioned, and what views are most likely to contain the information you seek.

Section 0 (“

- How a View is Documented") explains the standard organization used to document architectural views in this SAD. This tells you what section within a view you should read in order to find the information you seek.

1.1 Purpose and Scope of the SAD

This SAD specifies the software architecture for **<insert scope of SAD>.** All information regarding the software architecture may be found in this document, although much information is incorporated by reference to other documents.

What is software architecture? The software architecture for a system[1] is the structure or structures of that system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refers to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. This definition provides the basic litmus test for what information is included in this SAD, and what information is relegated to downstream documentation.

Elements and relationships. The software architecture first and foremost embodies information about how the elements relate to each other. This means that architecture specifically omits certain information about elements that does not pertain to their interaction. Thus, a software architecture is an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. Elements interact with each other by means of interfaces that partition details about an element into public and private parts. Software architecture is concerned with the public side of this division, and that will be documented in this SAD accordingly. On the other hand, private details of elements—details having to do solely with internal implementation—are not architectural and will not be documented in a SAD.

Multiple structures. The definition of software architecture makes it clear that systems can and do comprise more than one structure and that no one structure holds the irrefutable claim to being the architecture. The neurologist, the orthopedist, the hematologist, and the dermatologist all take a different perspective on the structure of a human body. Ophthalmologists, cardiologists, and podiatrists concentrate on subsystems. And the kinesiologist and psychiatrist are concerned with different aspects of the entire arrangement's behavior. Although these perspectives are pictured differently and have very different properties, all are inherently related; together they describe the architecture of the human body. So it is with software. Modern systems are more than complex enough to make it difficult to grasp them all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures. To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment—which *view* we are taking of the architecture. Thus, this SAD follows the

[1] Here, a system may refer to a system of systems.

principle that documenting a software architecture is a matter of documenting the relevant views and then documenting information that applies to more than one view.

For example, all non-trivial software systems are partitioned into implementation units; these units are given specific responsibilities, and are the basis of work assignments for programming teams. This kind of element will comprise programs and data that software in other implementation units can call or access, and programs and data that are private. In large projects, the elements will almost certainly be subdivided for assignment to sub-teams. This is one kind of structure often used to describe a system. It is a very static structure, in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.

Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The set of processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

None of these structures alone is *the* architecture, although they all convey architectural information. The architecture consists of these structures as well as many others. This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

These structures will be represented in the views of the software architecture that are provided in Section 3.

Behavior. Although software architecture tends to focus on structural information, *behavior of each element is part of the software architecture* insofar as that behavior can be observed or discerned from the point of view of another element. This behavior is what allows elements to interact with each other, which is clearly part of the software architecture and will be documented in the SAD as such. Behavior is documented in the element catalog of each view.

1.2 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 ("Documentation Roadmap") provides information about this document and its intended audience**. It provides the roadmap and document overview. Every reader who

wishes to find information relevant to the software architecture described in this document should begin by reading Section 1, which describes how the document is organized, which stakeholder viewpoints are represented, how stakeholders are expected to use it, and where information may be found. Section 1 also provides information about the views that are used by this SAD to communicate the software architecture.

- **Section 2 ("Architecture Background") explains why the architecture is what it is.** It provides a system overview, establishing the context and goals for the development. It describes the background and rationale for the software architecture. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been utilized in the architecture. It includes information about evaluation or validation performed on the architecture to provide assurance it meets its goals.
- **Section 3 (Views") and Section 4 ("Relations Among Views") specify the software architecture**. Views specify elements of software and the relationships between them. A view corresponds to a viewpoint (see Section 1.4), and is a representation of one or more structures present in the software (see Section 1.1).
- **Sections 6 ("Referenced Materials") and 7 ("Directory") provide reference information for the reader.** Section 5 provides look-up information for documents that are cited elsewhere in this SAD. Section 6 is a *directory*, which is an index of architectural elements and relations telling where each one is defined and used in this SAD. The section also includes a glossary and acronym list.

1.3 Stakeholder Representation

This section provides a list of the stakeholder roles considered in the development of the architecture described by this SAD. For each, the section lists the concerns that the stakeholder has that can be addressed by the information in this SAD.

Each stakeholder of a software system—customer, user, project manager, coder, analyst, tester, and so on—is concerned with different characteristics of the system that are affected by its software architecture. For example, the user is concerned that the system is reliable and available when needed; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried (in addition to cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways. The developer is worried about strategies to achieve all of those goals. The security analyst is concerned that the system will meet its information assurance requirements, and the performance analyst is similarly concerned with it satisfying real-time deadlines.

This information is represented as a matrix, where the rows list stakeholder roles, the columns list concerns, and a cell in the matrix contains an indication of how serious the concern is to a stakeholder in that role. This information is used to motivate the choice of viewpoints chosen in Section 1.4.

Stakeholder Role	Usability	Accuracy of Evaluations	Scalability	Reliability	System Accessibility	Cost	Maintainability	Performance	Schedule
Users	High	High	Medium	High	High	Low	Low	Medium	Low
Acquirers	Medium	Medium	High	Medium	Medium	High	High	Medium	High
Developers	Medium	High	High	High	High	Medium	High	High	High
Maintainers	Medium	Medium	High	High	Medium	Medium	High	Medium	Medium
Auditors	Low	High	Low	Medium	Medium	Low	Low	Low	Low
System and Software Integration Engineers	Low	High	High	High	Medium	Medium	High	High	Medium
Project Manager	Medium	Medium	Medium	Medium	Medium	High	Medium	Medium	High

1.4 Viewpoint Definitions

The SAD employs a stakeholder-focused, multiple view approach to architecture documentation, as required by ANSI/IEEE 1471-2000, the recommended best practice for documenting the architecture of software-intensive systems [IEEE 1471].

As described in Section 1.1, a software architecture comprises more than one software structure, each of which provides an engineering handle on different system qualities. A *view* is the

specification of one or more of these structures, and documenting a software architecture, then, is a matter of documenting the relevant views and then documenting information that applies to more than one view [Clements 2002].

ANSI/IEEE 1471-2000 provides guidance for choosing the best set of views to document, by bringing stakeholder interests to bear. It prescribes defining a set of viewpoints to satisfy the stakeholder community. A viewpoint identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view. A view, then, is a viewpoint applied to a system. It is a representation of a set of software elements, their properties, and the relationships among them that conform to a defining viewpoint. Together, the chosen set of views show the entire architecture and all of its relevant properties. A SAD contains the viewpoints, relevant views, and information that applies to more than one view to give a holistic description of the system.

The remainder of Section 1.5 defines the viewpoints used in this SAD. The following table summarizes the stakeholders in this project and the viewpoints that have been included to address their concerns.

Table 1: Stakeholders and Relevant Views

Stakeholder	View(s) that apply to that class of stakeholder's concerns
Developers	**Component-and-Connector View:** To understand runtime interactions between system components. **Decomposition View:** To grasp the modular breakdown and responsibilities of system components.
Maintainers	**Decomposition View:** To identify modules for future maintenance and updates. **Deployment View:** To understand where components are deployed for troubleshooting.
Project Managers	**Decomposition View:** To define work assignments and understand module responsibilities. **Deployment View:** To oversee resource allocation and system infrastructure.
System and Software Integration Engineers	**Component-and-Connector View:** To see how QualWeb integrates with the system. **Data Model View:** To verify how evaluation results are stored and accessed.

Stakeholder	View(s) that apply to that class of stakeholder's concerns
Auditors	**Data Model View:** To ensure compliance with WCAG standards and verify report integrity. **Deployment View:** To verify the security and reliability of the physical setup.

1.5 How a View is Documented

Section 3 of this SAD contains one view for each viewpoint listed in Section 1.5. Each view is documented as a set of view packets. A view packet is the smallest bundle of architectural documentation that might be given to an individual stakeholder.

Each view is documented as follows, where the letter *i* stands for the number of the view: 1, 2, etc.:

- Section 3.i: Name of view.
- Section 3.i.1: View description. This section describes the purpose and contents of the view. It should refer to (and match) the viewpoint description in Section 1.5 to which this view conforms.
- Section 3.i.2: Primary Presentation. Primary presentation. This section presents the elements and the relations among them that populate this view packet, using an appropriate language, languages, notation, or tool-based representation.
- Section 3.i.3: Element Catalog. Whereas the primary presentation shows the important elements and relations of the view packet, this section provides additional information needed to complete the architectural picture. It consists of subsections for (respectively) elements, relations, interfaces, behavior, and constraints.
- Section 3.i.4: Context Diagram. This section provides a context diagram showing the context of the part of the system represented by this view packet. It also designates the view packet's scope with a distinguished symbol, and shows interactions with external entities in the vocabulary of the view.
- Section 3.i.5: Architecture background. This section provides rationale for any significant design decisions whose scope is limited to this view packet.

2 Architecture Background

2.1 Problem Background

2.1.1 System Overview

Objectives

The system described is a Web Accessibility Monitoring Platform aimed at webmasters-individuals responsible for managing and maintaining websites. Its purpose is to ensure that websites comply with accessibility standards, particularly those outlined by the Web Content Accessibility Guidelines (WCAG).

Functionalities and Main features

The system serves as a centralized platform where webmasters can:

1. Register Websites and Pages:
    - Input the URLs of websites and specific pages to be monitored.
    - Validate URLs and associate them with the respective domains.
2. Evaluate Accessibility:
    - Use an automated tool (QualWeb Core) to perform accessibility checks on pages. Identify errors, warnings, and successful compliance with WCAG standards.
3. Visualize Results:
    - Provide a dashboard for tracking the status of websites and pages.
    - Display detailed results of evaluations, including compliance percentages and issue breakdowns.
4. Aggregate Accessibility Indicators:
- Summarize metrics, such as the percentage of conformant and non-conformant pages.
- Highlight common accessibility errors across pages.
5. Generate Reports:
- Create reports in HTML formats for compliance or documentation purposes.

6. Manage and Maintain Data:

- Allow webmasters to delete outdated or irrelevant data, such as websites or pages, ensuring the system remains organized and relevant.

2.1.2 Goals and Context

The primary goal of this software is to empower webmasters with tools to ensure their websites comply with accessibility standards, particularly the **Web Content Accessibility Guidelines (WCAG)**.

Role of Software Architecture

The software architecture establishes the foundation for achieving the system's objectives by ensuring:

1. **Scalability**: The system must manage an increasing number of websites and pages as webmasters expand their monitoring needs.

2. **Accessibility**: Aligning with WCAG standards is both a goal of the system and a requirement for the platform itself, as it must be fully navigable using keyboards and screen readers.

3. **Modularity**: A clear separation of concerns, implemented through a layered architecture, allows for independent development and maintenance of the system's core functionalities (interface, accessibility evaluation, reports, etc.).

4. **Performance**: By leveraging modern web technologies and efficient data handling, the system supports fast accessibility evaluations and smooth user interaction, even with large datasets.

5. **Reliability**: The system's architecture ensures that evaluations, data storage, and reporting are robust and error-resilient, essential for maintaining user trust.

2.1.3 Significant Driving Requirements

Quality Attribute Requirements

The quality attribute scenarios (QAS) are listed below, separated by quality attribute.

Usability

- QAS1. A new user enters the system to add a new website, it should be able to complete the process of entering the URL and adding at least one page to monitor without needing any additional instructions or support. The task should take no more than 2 minutes.
- QAS2. A user wants to interact with the platform using only the keyboard. The user must be able to navigate through all the processes with minor inconvenience and with a speed comparable to using the mouse.

Modifiability

- QAS3. The system needs to be updated to accommodate the introduction of new accessibility standards. The new standards must be integrated with no more than 1 person-days of effort, these changes should be implemented with no more than 30 minutes of downtime for users and the system must remain fully functional after the update.
- QAS4. The system must be able to replace its current accessibility analyzer (QualWeb) with a new accessibility evaluation platform, while maintaining the overall functionality and integrity of the system. The time to complete the replacement should not exceed 3 person-days, and the evaluation results must be stored in the database with the same structure as before to be compatible with older evaluation results.

Performance

- QAS5. A user starts an accessibility evaluation of a website with up to 100 pages. The system completes the evaluation of all pages in under 5 minutes and provides the evaluation results within 10 seconds after completion.
- QAS6. Up to 10 users start an evaluation of a website containing up to 100 pages each following a random distribution over 1 minute, the system should process the evaluation requests without major degrading performance, with an average response time under 10 minutes for each website evaluation.

Availability

- QAS7. Websites and pages added to the platform must remain in the database persistently, as well as their accessibility evaluation results, and must be available for retrieval at any time, unless explicitly deleted by the user.

2.2 Solution Background

2.2.1 Architectural Approaches

The architecture of the platform was designed to balance the behavioral requirements (e.g., functional capabilities) and quality attributes (e.g., accessibility, scalability, and modifiability). The following architectural approaches underpin the system's design:

Architectural Styles and Patterns

1. **Layered Architecture**
    a. **Description**: The platform is built using a four-layer architecture:
        1. **Client Browser Layer**: Angular-based user interface implementing Material Design for accessibility.
        2. **Web Server Layer**: A Node.js server handling user requests and routing them appropriately.
        3. **Application Server Layer**: A separate Node.js instance for managing business logic, such as triggering accessibility evaluations and aggregating results.
        4. **Database Layer**: MongoDB for structured storage of websites, pages, evaluations, and reports.
    b. **Rationale**:
        - This separation of concerns ensures that each layer can be independently developed, tested, and maintained.
        - The clear division allows for scalability and adaptability to future changes or extensions.
2. **RESTful Architecture**
    a. **Description**: The system employs RESTful APIs to communicate between the client and server layers.
    b. **Rationale**:

- RESTful APIs are stateless and lightweight, making them ideal for web applications with high scalability and reliability needs.

- Standardized communication simplifies integration with external tools like QualWeb Core and potential future services.

3. **Use of Design System (Material Design)**

    a. **Description**: Material Design principles guide the UI/UX development, focusing on consistency, responsiveness, and accessibility.

    b. **Rationale**:

        - Material Design ensures a user-friendly and WCAG-compliant interface, meeting the needs of diverse user groups, including those with disabilities.

4. **Event-Driven Architecture for Evaluations**

    a. **Description**: Accessibility evaluations are initiated asynchronously, with events managing the state of the evaluations ("In evaluation", "Evaluated").

    b. **Rationale**:

        - Event-driven approaches allow for efficient handling of concurrent evaluation tasks. (Important when a substantial number of users are using the platform!)

        - Decoupling evaluation processes from the user interface enhances responsiveness and performance.

Technological Frameworks and Tools

1. **MEAN Stack**

    a. **Description**: The system uses MongoDB, Express.js, Angular, and Node.js for its implementation.

    b. **Rationale**:

- The MEAN stack is a cohesive, JavaScript-based framework that supports rapid development and easy integration between layers.
- MongoDB's NoSQL structure provides flexibility in storing nested, evolving data such as evaluation results.

2. **Integration with QualWeb**

    a. **Description**: QualWeb was the chosen tool for accessibility evaluations.

    b. **Rationale**:

        i. QualWeb is a robust, standards-compliant tool that automates accessibility checks, ensuring precise results aligned with WCAG guidelines.

        ii. Its compatibility with Node.js simplifies integration within the application server.

Alternatives Considered

1. **Monolithic Architecture**: A single-tier monolithic approach was deemed unsuitable as it would limit scalability and make maintenance more challenging.

COTS (Commercial Off-The-Shelf) Issues

1. **QualWeb Core:**

    a. **Issue**: Dependency on a third-party tool introduces risks, such as potential changes in its API or licensing.

    b. **Mitigation**: Regular updates and fallback mechanisms ensure compatibility and resilience.

2. **Material Design Components:**

a. **Issue**: Strict adherence to Material Design may limit customization options for specific accessibility needs.

b. **Mitigation**: Custom overrides ensure the platform remains WCAG-compliant while meeting user requirements.

2.2.2 Analysis Results

Qualitative Analysis

1. **Alignment with Quality Attributes**

    a. **Accessibility**:

        i. The use of Material Design ensures compliance with WCAG standards for the platform itself.

        ii. The ability to navigate using only a keyboard was verified in the UI of the platform, ensuring accessibility for the users.

    b. **Scalability**:

        i. The layered architecture supports horizontal scaling by allowing the application server and database to expand independently if the user base grows.

        ii. RESTful APIs ensure stateless interactions, minimizing bottlenecks during data exchanges.

    c. **Modifiability**:

        i. The modular structure facilitates changes, such as integrating additional functionalities.

    d. **Performance**:

        i. Asynchronous processing for accessibility evaluations allows the system to manage multiple concurrent tasks efficiently.

2. **Risk Mitigation**

a. **Dependency Risks**: Maintaining QualWeb updated regularly or using alternative accessibility evaluation tools.

b. **Technology Risks**: The use of the MEAN stack minimizes risks by leveraging widely adopted technologies with strong community support.

Quantitative Analysis

1. Performance Modeling

a. Simulated tests of RESTful API interactions were conducted to evaluate system response times:

i. Results showed average response times of <500ms for most API calls.

ii. Multiple evaluation tasks processed simultaneously without majorly affecting the user interface's responsiveness.

2. Accessibility Testing

a. Manual testing confirmed that the design supports screen readers and keyboard navigation.

3. Architecture Tradeoff Analysis Method (ATAM) Highlights

a. **Tradeoffs between performance and modifiability**: The choice of a layered architecture slightly increases response time due to additional abstraction layers but provides significant benefits for maintainability and scalability.

b. **Balancing accessibility and customizability**: Strict adherence to Material Design limits UI flexibility but ensures high accessibility compliance.

3 Views

This section contains the views of the software architecture. A view is a representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. Concretely, a view shows a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.

Architectural views can be divided into three groups, depending on the broad nature of the elements they show. These are:

- Module views. Here, the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. Modules are assigned areas of functional responsibility, and are assigned to teams for implementation. There is less emphasis on how the resulting software manifests itself at runtime. Module structures allow us to answer questions such as: What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use? What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

- Component-and-connector views. Here, the elements are runtime components (which are principal units of computation) and connectors (which are the communication vehicles among components). Component and connector structures help answer questions such as: What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes?

- Allocation views. These views show the relationship between the software elements and elements in one or more external environments in which the software is created and executed. Allocation structures answer questions such as: What processor does each software element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of the software element to development teams?

These three kinds of structures correspond to the three broad kinds of decisions that architectural design involves:

- How is the system to be structured as a set of code units (modules)

- How is the system to be structured as a set of elements that have run-time behavior (components) and interactions (connectors) ?

- How is the system to relate to non-software structures in its environment (such as CPUs, file systems, networks, development teams, etc.)?

Often, a view shows information from more than one of these categories. However, unless chosen carefully, the information in such a hybrid view can be confusing and not well understood.

The views presented in this SAD are the following:

Name of view	Viewtype that defines this view	Types of elements and relations shown	Is this a module view?	Is this a component-and-connector view?	Is this an allocation view?
Data Model View	Module View	Data entities (Website, Page, Report) and their relationships (Website has Pages, Page has Report).	**Yes**	No	No
Component-and-Connector (C&C) View	Component-and-Connector View	Runtime components (Client Browser, Web Server, Application Server, Database, QualWeb) and their interactions (HTTP requests, REST APIs).	No	**Yes**	No
Decomposition View	Module View	Logical modules (Frontend, Backend, Controllers, Models, Routes) and their hierarchical breakdown.	**Yes**	No	No
Deployment View	Allocation View	Physical nodes (Client Machine, Web Server, Application Server, Database Server) and the deployment of components onto these nodes.	No	No	**Yes**

3.1 Data Model View

3.1.1 View Description

This view describes the static information structure of the system in terms of data entities and their relationships. It provides a detailed view of the data model, which includes the key entities, their attributes, and the relationships between them.

3.1.2 Primary Presentation

3.1.3 Element Catalog

- **Website**: Represents a website with attributes such as URL, validation status, pages, registration date, and last validation date.

- **Page**: Represents a page within a website with attributes such as URL, validation status, report, registration date, and last validation date.

- **Report**: Represents the result of the accessibility evaluation done about the respective page.

3.1.4 Architecture background

Domain Driven Design:

- The model reflects the main concepts of a Web Accessibility Monitoring Platform.

- Entities like Website, Page, and Report map directly to domain objects.

Scalability:

- This structure ensures that the system can handle a growing number of Websites and their associated Pages.
- The design supports, for example, the addition of new attributes to the Report without significant changes to other entities.

Compliance with Standards:

- The Report entity is designed to align with Web Content Accessibility Guidelines, ensuring that the evaluations capture all relevant metrics (errors, warnings, etc.).

Flexibility:

- This model allows for changes and adaptations, for example, to use another evaluation tool, allowing the model to adapt to new requirements or future versions.

Database:

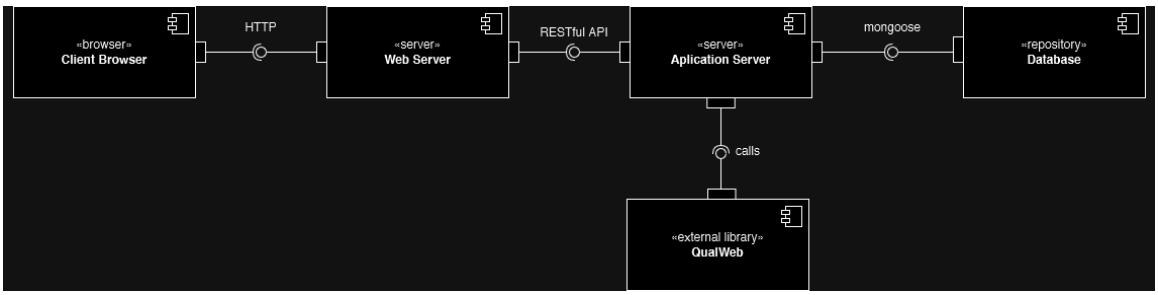
- The MongoDB database choice influenced this architecture. The way entities like Website and Page are stored in the database makes queries fast and efficient while maintaining a logical structure.

3.2 Component-and-Connector (C&C) View

3.2.1 View Description

This view describes the major components of the system and how they interact with each other through connectors. It focuses on the runtime interactions between the components to fulfill the system's functionality.

3.2.2 Primary Presentation



3.2.3 Element Catalog

- **Client Browser:** Represents the user interface of the application running on a web browser. It is responsible for displaying the user interface, handling user interactions, and communicating with the web server. The client browser sends HTTP requests to the web server to load the application and make API calls. It receives responses and updates the user interface accordingly.

- **Web Server:** Handles HTTP requests from the client browser. It serves static files to the client browser and forwards API requests to the application server. The web server acts as an intermediary between the client browser and the application server, ensuring that requests are properly routed and responses are sent back to the client.

- **Application Server**: Processes API requests received from the web server, executes the necessary business logic, and interacts with the database to retrieve or update data. The application server also communicates with the QualWeb evaluator to perform accessibility evaluations.

- **Database:** Responsible for storing and retrieving data as requested by the application server. The database server ensures data integrity and provides efficient data access mechanisms. It handles queries and updates the database, supporting the application's data needs.

- **QualWeb**: External library used to evaluate the accessibility of web pages. The application server sends evaluation requests to the QualWeb evaluator, which performs the accessibility evaluations and returns reports.

3.2.4 Architecture background

Layered Architecture Style:

The system follows a layered style for clear separation of concerns:

- Client Layer (Client Browser) handles user interaction and UI rendering.
- Presentation Layer (Web Server) routes requests.
- Business Logic Layer (Application Server) executes application logic.
- Data Layer (Database) manages persistent storage.

Modularity:

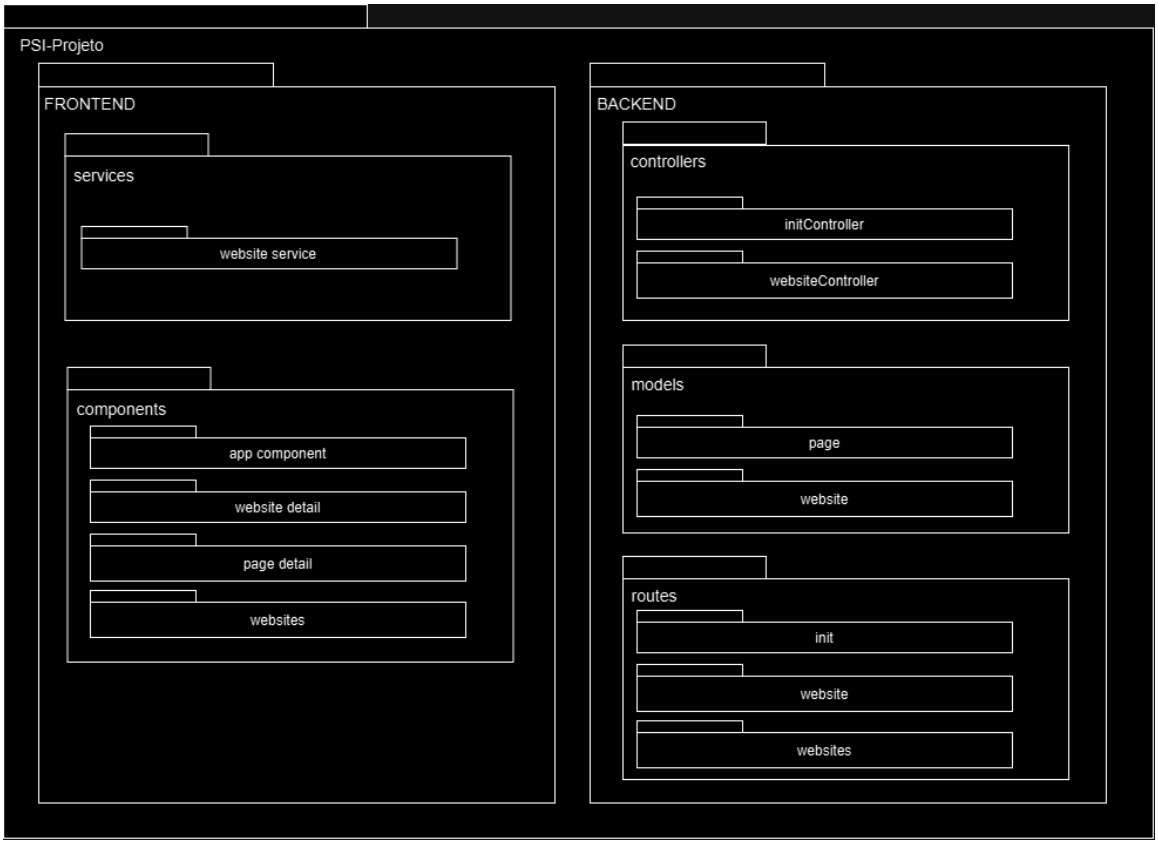
- Each component is modular, making maintenance, testing, and scaling easier. For example, the QualWeb Evaluator is an independent module that can be replaced or updated without affecting the other components.

3.3 Decomposition View

3.3.1 View Description

This view shows the hierarchical structure of the project, breaking it down into modules and sub-modules. This view helps in understanding the organization of the codebase and the responsibilities of each module.

3.3.2 Primary Presentation



3.3.3 Element Catalog

- **PSI-Projeto:** Represents the system.
- **BACKEND**: Responsible for handling server-side logic, database interactions, and API routes. It processes incoming HTTP requests, interacts with the database, and sends responses back to the FRONTEND.
- **Controllers**: Handles incoming HTTP requests, process them, and return appropriate responses. They act as intermediaries between the client and the backend services.
    - **InitController**: Handles initialization logic.
    - **websiteController**: Manages website-related operations.
- **Models**: Defines the structure of the data and interact with the database. They represent the application's data and provide methods to query and manipulate it.

    - **Page**: Defines the schema for a page.
    - **Website**: Defines the schema for a website.

- **Routes**: Defines the endpoints for the API and map them to the corresponding controller functions. They handle the routing of HTTP requests to the appropriate controllers.
    - **Init**: Routes for initialization.
    - **Website**: Routes for individual website operations.
    - **Websites**: Routes for multiple websites operations.

- **FRONTEND**: Responsible for handling client-side logic, user interface, and interactions with the BACKEND via HTTP requests. It makes HTTP requests to the BACKEND to fetch or manipulate data and updates the UI based on the responses received

- **Services**: Handles business logic and data fetching/manipulation. They interact with the backend API to fetch or update data and provide it to the components.
    - **Website Service**: Service for interacting with the backend to fetch and manipulate website data.

- **Components**: Components are the building blocks of the user interface. They encapsulate the HTML, CSS, and TypeScript code necessary to render a part of the UI and manage its behavior.
    - **App Component**: Main application component.
    - **Website Component**: Component for displaying a list of websites.
    - **Website Details Component**: Component for displaying details of a specific website.
    - **Page Details Component**: Component for displaying details of a specific page.

3.3.4 Architecture background

This system adopts a modular architecture, splitting responsibilities between the Frontend and Backend for clarity, maintainability, and scalability:

1. Frontend:
    - Designed using a component-based architecture, where each UI component is self-contained, focusing on specific functionality.
    - Services handle communication with the backend, ensuring separation of concerns between UI rendering and data handling.
2. Backend:
    - Follows the Model-View-Controller (MVC) pattern:
        - Models define data schemas and handle database operations.
        - Controllers manage application logic and act as intermediaries between Routes and Models.
        - Routes define API endpoints and ensure requests are routed to the appropriate controllers.
3. Separation of Responsibilities:
    - This architecture ensures the frontend handles user interactions and the backend processes business logic and manages data.
    - Communication between the two layers is facilitated by RESTful APIs.

3.4 Deployment View

3.4.1 View Description

This view illustrates the physical deployment of software components on hardware nodes within the system. It provides a high-level overview of how the different parts of the application are distributed across various servers and devices. This view helps in understanding the physical distribution of the software components, their interactions, and the communication pathways between them.

3.4.2 Primary Presentation

3.4.3 Element Catalog

- **Client Machine**: The user's physical or virtual device that runs the Client Browser. It acts as the interface where the user accesses the Web Application through a web browser. The Client Machine interacts with the Web Server over HTTP requests initiated from the Client Browser.

- **Web Server**: Responsible for hosting the Web Application and processing HTTP requests from the Client Machine. The Web Server forwards requests to the Application Server to handle business logic.

- **Web Application**: Part of the Web Server that contains the web-based frontend logic. The Web Application is responsible for rendering content in the browser and interacting with the backend services. The Web Application is an Angular framework. It is hosted on the Web Server and interacts with the Application Server via REST APIs.

- **Application Server**: A server that hosts the backend application logic. It processes requests received from the Web Server, performs business logic, and interacts with the Database Server to retrieve and store data. The Application Server uses Node.js. It also integrates with QualWeb through REST APIs.

- **Database Server**: Responsible for storing and managing the application's data. It communicates with the Application Server to retrieve, modify, and store data. The Database Server is a MongoDB database.

- **QualWeb**: Integrates with the Application Server to evaluate web pages' accessibility and generate reports, providing insights on improving accessibility across the web application.

3.4.4 Architecture background

This architecture uses a layered deployment to ensure scalability, reliability, and clear separation of responsibilities:

1. **Separation of Concerns**:
   Each component handles a specific responsibility:
   - Web Server focuses on delivering the frontend.
   - Application Server handles backend logic.
   - Database Server manages data storage.
2. **Scalability**:
   Components are deployed on different servers, allowing independent scaling based on load.
3. **Efficiency with REST APIs**:
   REST APIs enable flexible, stateless communication between components, making the system modular and easier to maintain or update.

This design ensures the system is future-proof, adaptable, and inclusive, meeting both functional and accessibility requirements.

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the views give different system perspectives, they are not independent. Elements of one view will be related to elements of other views, and we need to reason about these relations. For example, a module in a decomposition view may be manifested as one, part of one, or several components in one of the component-and-connector views, reflecting its runtime alter-ego. In general, mappings between views are many to many. Section 4 describes the relations that exist among the views given in Section 3. As required by ANSI/IEEE 1471-2000, it also describes any known inconsistencies among the views.

Data Model ↔ C&C View

Element in Data Model View	Corresponding Element in C&C View
Website	Application Server → Database
Website	Application Server → Database
Report	Application Server → QualWeb → Database

C&C ↔ Deployment View

Element in C&C View	Corresponding Element in Deployment View
Client Browser	Client Machine
Web Server	Web Server
Application Server	Application Server
Database	Database Server

5 Code analysis and Quality Enhancement

5.1 Tools Used for Code Analysis

We used the SonarQube platform to analyze the project's source code. SonarQube provides a suite of tools for detecting code quality issues, measuring maintainability, and ensuring adherence to the best practices. Metrics such as code smells, bugs, vulnerabilities, test coverage, and duplication are presented.

5.2 Summary of Detected Issues

Category	Main Issues Detected	Quality Attributes Affected
Security	MongoDB database username and password present on the code.	This can expose the system to attacks, such as information leaks or unauthorized access.
Maintainability	Mainly, use of var instead of let/const. Other less impactful code issues.	Using var declares function-scoped or global-scoped variables. Using let declares block-scoped variables, and const for variables that are constant, which helps maintainability.
Coverage	Critical modules lack test cases.	As we did not implement test cases for critical modules (we only tested manually), this reduces confidence in the system's correctness and makes it harder to detect regressions.
Duplication	Some duplicated code in a module.	Duplicated code increases the effort required for updates and introduces some risk of inconsistent behavior. It also makes maintainability even harder.

5.3 Issue Fixing and Improvements

The following changes were made to the project's codebase to address issues identified during the analysis:

1. **Improving Security: Use of dotenv for Environment Variables**

- **Issue:** Hardcoded credentials (username and password) were detected, posing a security vulnerability.
- **Fix:**
    a. Integrated the dotenv package in our project.
    b. Added the .env file to .gitignore to prevent this sensitive data from being included in version control.
    c. Updated the code to use environment variables for the MongoDB connection string, preventing the username and password from showing in the code.

2. **Refactoring Variable Declarations**

- **Issue:** Several variables were declared using var, leading to harder maintainability and potentially other issues.
- **Fix:** Replaced var with let or const as appropriate, adhering to modern JavaScript best practices.

3. **Simplifying Regular Expressions**

- **Issue:** Redundant complexity was identified in regular expressions used for URL validation.
- **Fix:** Removed unnecessary characters in the regular expression to make it simpler and more readable without affecting its functionality.

5.4 Effects of Changes in Metrics

Metric	Before Fixes	After fixes
Security	2 issues	0 issues
Maintainability	49 issues	2 issues
Security Hotspots	6 issues	0 issues

5.5 Lessons Learned

With this analysis, we learned the value of continuous code analysis. Integrating tools like SonarQube into the development process would have helped us identify issues early, making us more efficient and our software more robust and secure.

Addressing code smells like duplication, not only improves maintainability, but also makes the system easier to test and extend.

Fixing vulnerabilities like the hardcoded username and password of our database also massively improves security of our software and highlights the importance of secure code practices.

6 Referenced Materials

Barbacci 2003	Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs)*, Third Edition (CMU/SEI-2003-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr016.html>.
Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice,* second edition, Addison Wesley Longman, 2003.
Clements 2001	Clements, Kazman, Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley Longman, 2001.
Clements 2002	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.

7 Directory

7.1 Glossary

Term	Definition
software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
view	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
view packet	The smallest package of architectural documentation that could usefully be given to a stakeholder. The documentation of a view is composed of one or more view packets.
viewpoint	A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and audience for a view, and the techniques for its creation and analysis [IEEE 1471]. Identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view.

QualWeb	Accessibility evaluation tool
MEAN stack	MongoDB, Express.js, Angular, Node.js

7.2 Acronym List

API	Application Programming Interface; Application Program Interface; Application Programmer Interface
ATAM	Architecture Tradeoff Analysis Method
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
CORBA	Common object request broker architecture
COTS	Commercial-Off-The-Shelf
EPIC	Evolutionary Process for Integrating COTS-Based Systems
IEEE	Institute of Electrical and Electronics Engineers
KPA	Key Process Area
OO	Object Oriented
ORB	Object Request Broker
OS	Operating System
QAW	Quality Attribute Workshop
RUP	Rational Unified Process
SAD	Software Architecture Document
SDE	Software Development Environment
SEE	Software Engineering Environment
SEI	Software Engineering Institute Systems Engineering & Integration Software End Item
SEPG	Software Engineering Process Group
SLOC	Source Lines of Code
SW-CMM	Capability Maturity Model for Software
CMMI-SW	Capability Maturity Model Integrated - includes Software Engineering
UML	Unified Modeling Language
WCAG	Web Content Accessibility Guidelines