Architecture document

**Table of Contents**

1. Introduction
2. System Context (C1)
3. Containers and tech choices (C2)
4. Components (C3)
5. UML

**Introduction**

This document aims to describe the software architecture, delving into its functioning and the rationale behind the design decisions. It provides a detailed walkthrough of the C4 model, offering insights into each segment.

**System Context (C1)**

**A diagram of a software application

Description automatically generated**

**Containers and tech choices (C2)**

A diagram of software development

Description automatically generated

The Ordina Website platform is built upon **three primary tiers** – Front End (FE), Back End (BE), and Database (DB). **Here are a few reasons for that**:

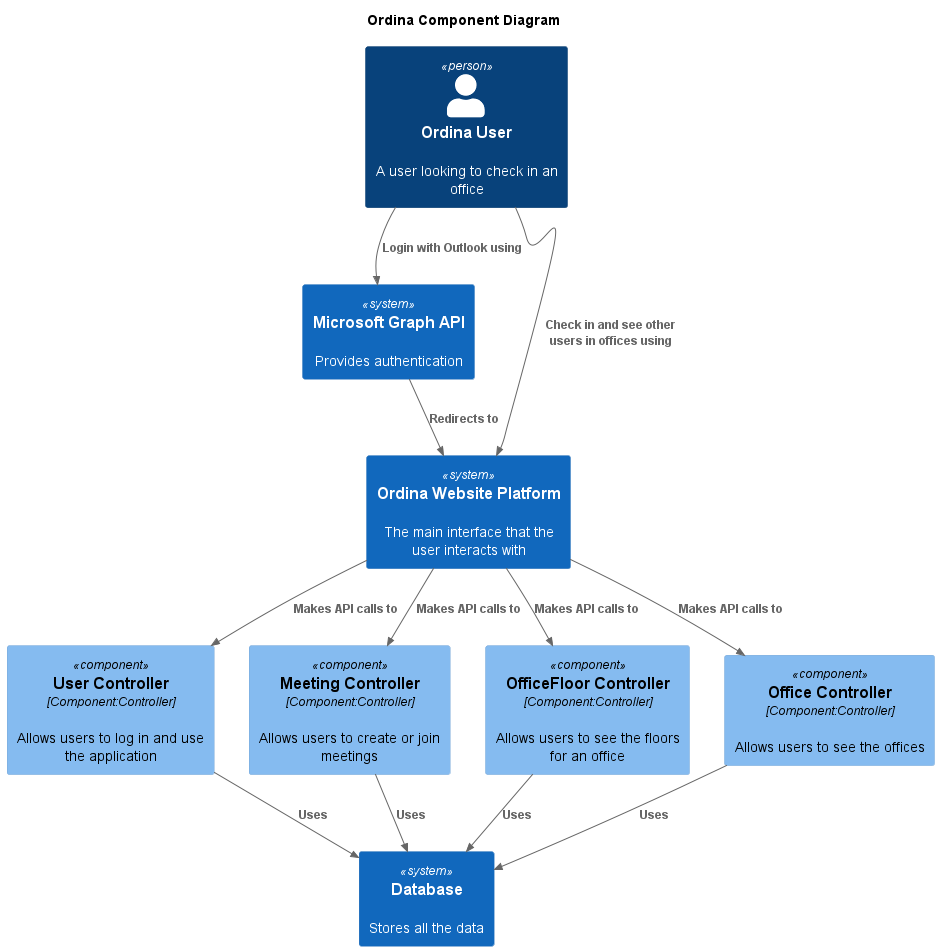
1. It gives you the ability to update the technology stack of one tier, without impacting other areas of the application. For example, if we need to change from React to Angular the Backend and the Database will still work for the new frontend.
2. You can scale the application up and out. A separate back-end tier, for example, allows you to deploy to a variety of databases instead of being locked into one particular technology. It also allows you to scale up by adding multiple web servers.
3. It adds reliability and more independence of the underlying servers or services.
4. It provides an ease of maintenance of the code base, managing presentation code and business logic separately, so that a change to business logic, for example, does not impact the presentation layer.

For the FE, we opted for React. This decision was influenced by React's component-based code organization, which enhances readability and simplifies the process when dealing with numerous files. Additionally, the vast selection of libraries available and the supportive community further enrich its development capabilities. React's beginner-friendly attributes also played a role in this choice.

Opting for Node.js as the API layer in a React-based application offers several benefits. It ensures a cohesive JavaScript environment, simplifying development across the front and back ends. Node.js is known for its impressive performance, thanks to non-blocking I/O and asynchronous processing, which is particularly effective for high-traffic and real-time applications. The platform also enjoys robust community support, providing an abundance of resources and libraries, similar to React.

Choosing MongoDB as the database for a Node.js and React project offers numerous advantages. Its JSON-like data format (BSON) aligns seamlessly with the JavaScript ecosystem, simplifying data handling and manipulation across both Node.js and React. MongoDB is known for its scalability and flexibility, capable of handling large volumes of data and high throughput, which is essential for growing applications. It also provides high performance in read and write operations, crucial for real-time data processing, and features a rich query language for complex data operations. This compatibility extends to the MERN stack (MongoDB, Express.js, React, Node.js), a widely adopted industry standard that facilitates efficient full-stack development. Additionally, MongoDB's strong community support ensures a wealth of resources and continuous tool and extension development, enhancing its functionality and ease of use in diverse applications.

Components **(C3)**



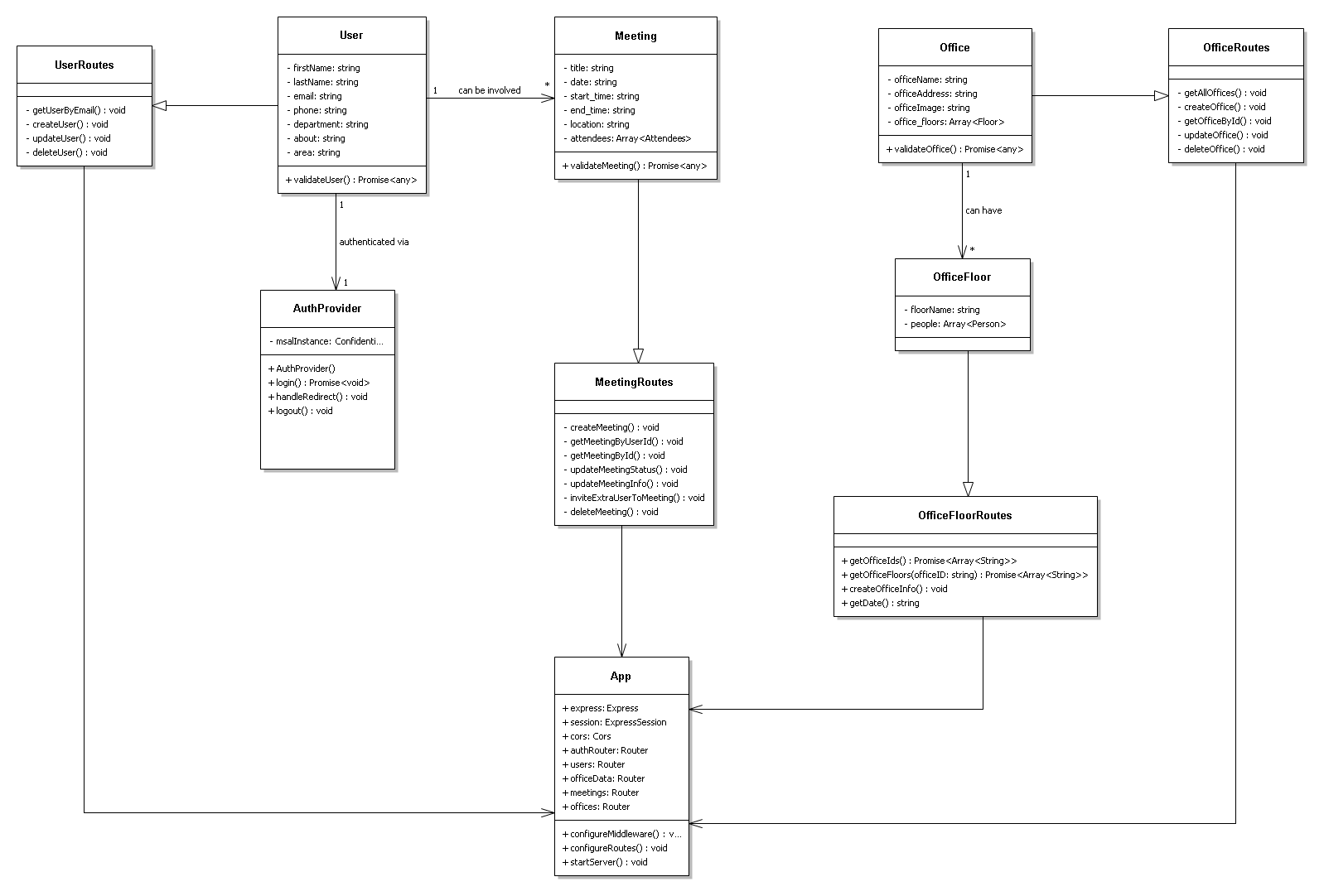
The architecture of the Ordina system is composed of several key components that interact with each other to provide a comprehensive service to the end-users. These components are integrated into a larger system, which includes both front-end and back-end elements, and a database to store all the relevant data.

The system begins with the end-user, who interacts with the Microsoft Graph API for authentication purposes. This interaction enables the user to log in and access the Ordina website platform. The website platform serves as the main interface for the user, providing various functionalities such as checking in at an office and interacting with other users in the offices.

The website platform communicates with four different components: the User Controller, Meeting Controller, OfficeFloor Controller, and Office Controller. Each of these controllers handles specific functionalities within the system. The User Controller manages user login and usage of the application, the Meeting Controller handles meeting creation and joining, the OfficeFloor Controller allows users to view the floors of an office, and the Office Controller enables users to view different offices.

All these components interact with the database, which stores all the data required by the application. This separation of concerns allows for easy updates and scaling of individual components without affecting the overall system. Furthermore, this modular approach enhances the reliability of the system and simplifies maintenance efforts.

UML **(C4)**



This UML diagram helps to illustrate the structure of the system and the interactions between its different components. It also shows how the system uses different technologies (React, Node.js, MongoDB) to implement its functionalities.

In the context of the Ordina Website platform, the Single Responsibility Principle (SRP) is evident in the division of responsibilities among different components of the system. Each component is designed to handle a specific functionality within the system.

For example:

* The UserController manages user login and usage of the application.
* The MeetingController handles meeting creation and joining.
* The OfficeFloorController allows users to view the floors of an office.
* The OfficeController enables users to view different offices.

Each of these controllers has a single responsibility, and hence, a single reason to change. If a change is needed in the way user login is handled, it would only require a change in the UserController. Similarly, if there's a change in how meetings are created, it would only require a change in the MeetingController.

This adherence to the SRP makes the system more maintainable and easier to understand. Changes in one component do not affect others, reducing the risk of introducing bugs. It also improves the reusability of the code, as each component can be used independently in different parts of the system.