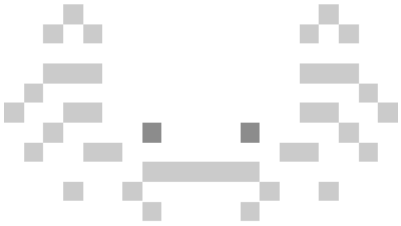


Software name:	SpiderByte
Project title:	Project 1: Communication application
Client name:	Professional Services, Department of Informatics
Team members:	Munkhtulga Battogtokh Adriel Aiach-Kohen Aakash Doshi Alin Fulga Kymani Lawrence Maiwand Maidanwal Dimitris Papatheodoulou Mohammad Sadi Henry Valeyre

Contents

Introduction

"The primary objective of the project is to produce an application that enables the department to communicate *more effectively* with students directly by sending messages and announcements to their mobile phone, from a web-based administrator's interface".

In other words, client wants an application that makes department-to-student one-way communication *easier, better organised*, *and better controlled* for both the **admins** who create the communication-content (announcement), and the **students** who receive them.

The system-as-is, without our application, is that our client (Professional Services members—admins) sends announcement, and event information through email. This involves meticulous process of having Student Records produce list of students on certain filter-criteria, and manually copying email addresses to send content. Considering this among other factors, we agree with client that the objectives of this project are to achieve the respective for the *main stakeholders:*

Admins (client, users)	Ease of spreading information from dedicated Web User Interface, flexible and easy way to target the right group, save administrative work
Students (users)	Up-to-date information, all Department announcements organised in one mobile app, avoiding of irrelevant data, access to information
Team (developers)	Practice team-work, learn new technologies

Outcome

What have we built, and achieved?

SpiderByte is server-client application consisting of three smaller applications: Web server written in NodeJS, and Android & iOS mobile applications written in Kotlin, and Swift respectively. At the heart, our application is for communication between the Web, and the Mobile. One can think of iOS, and Android as two versions of the same app, and in fact, we have actively tried to make them as similar as possible. Also, we put effort in integrating the current practices, and systems employed by our client into SpiderByte, so to save administrative work, and make our apps as useful as they could be.

In summary, we've achieved all objectives by each stakeholder as given in *Introduction.* Namely:

For the admins:

- Simple, tailored, secure, and yet flexible Web UI for managing announcements, and events.
- Easier, powerful, and flexible procedure to target certain student groups, which is integrated with the current system, and practices employed by our client.
- No need to do any administrative work managing students (registering, deleting etc.).

For the students:

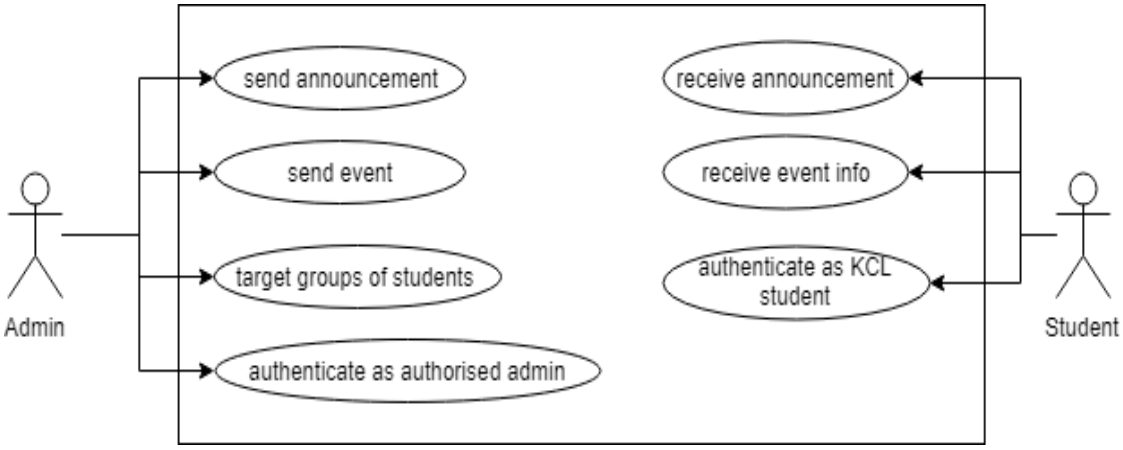
- Real-time communication between Web, and Mobile apps for "up-to-date information"
- Mobile apps on the biggest two platforms to give wide student population access to SpiderByte.

For the team:

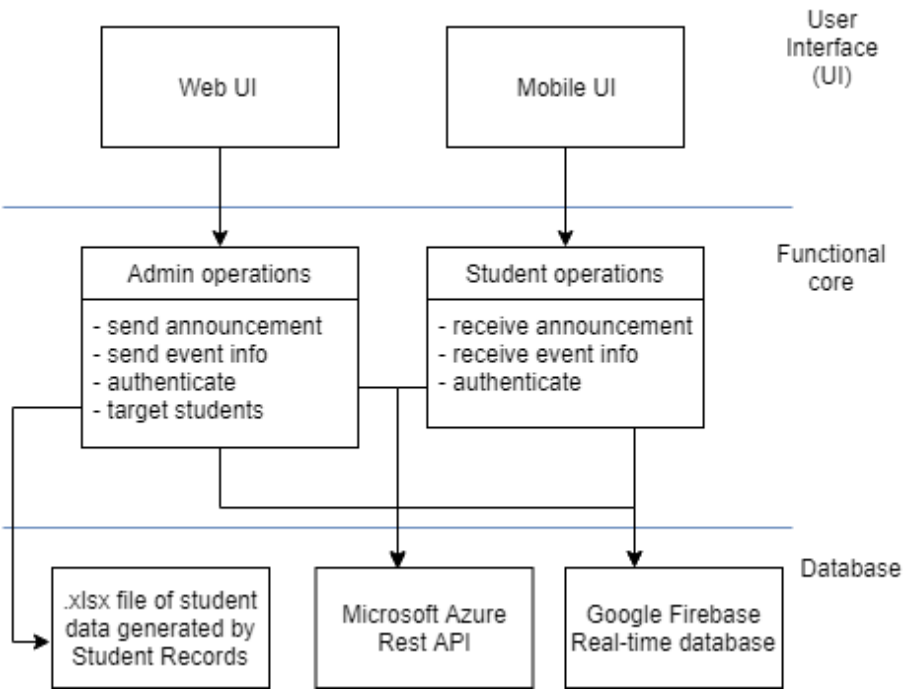
- New programming languages, team-work experience, improved efficiency.

How does SpiderByte achieve the objectives?

Below is Use Case Diagram illustrating the objectives as use-cases. These are the first step into reducing the objectives into technological problems.



Below is architecture diagram illustrating how we accommodate each of the use case above in SpiderByte.



To clarify elements of the architecture diagram, we simply divide the admin, and student operations to the Web, and Mobile platforms respectively. Furthermore, since we can't get access to Student Records' own database, we must retrieve student data from "xlsx" format files that Student Records generate (This is the current practice our client employs). More details in the next section.

What have we employed to make the achievements true (dependencies, APIs, libraries)?

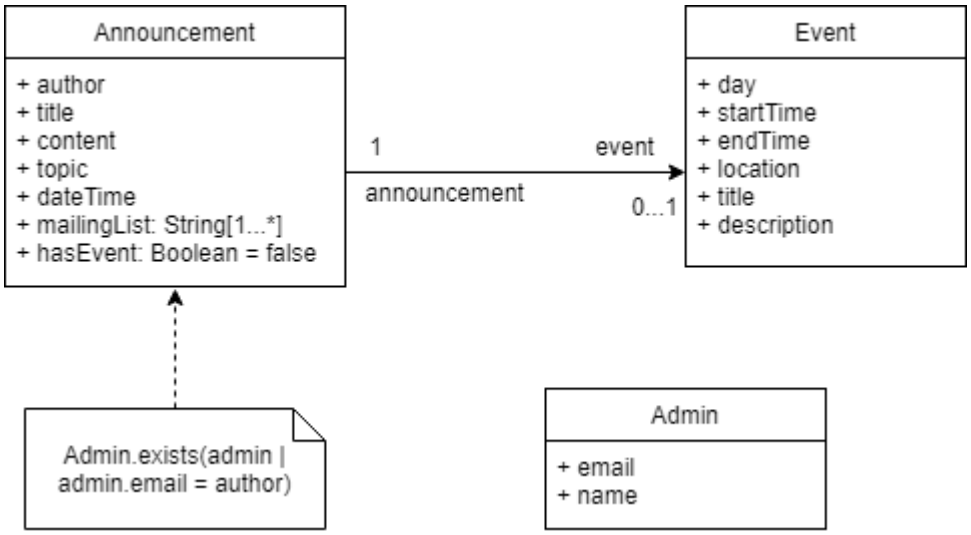
As shown on architecture diagram, SpiderByte is integrated with Microsoft Azure Rest API for both the student, and admin data. This is essential in achieving the objective of saving administrative work, and authentication of both students, and admins.

On the other hand, Firebase is used to store announcement, event, and some admin related information. This is of course important for the main objective of web-mobile communication. Next page contains the list for all other dependencies by each platform (each platform subdirectory on Github Enterprise contains their own Readme.MD file listing these, as well as the licenses):

iOS	Web		Android
Microsoft Authentication Library (MSAL) JTAppleCalendar Paper-onboarding Floaty HTagView Down Firebase-ios-sdk OC Mock FontAwesomeIcons	Node-js Body-parser chai-as-promised chai-jquery compression cookie-parser date-time debug express express-session express-validator firebase-admin firebase-functions googleapis helmet install jade jsdom markdown markdown-to-html	mocha-retry morgan multer node-datetime node-schedule node-xlsx nodemailer npm passport passport-azure-ad selenium-webdriver serve-favicon sleep xlsx xlsx-style chai chai-http nodemon bootstrap imgbb	Microsoft Graph API MarkdownView Material-Calendar-View AndroidTagGroup AndroidPdfViewer FloatingActionButton Applandeo

Design

The domain model:



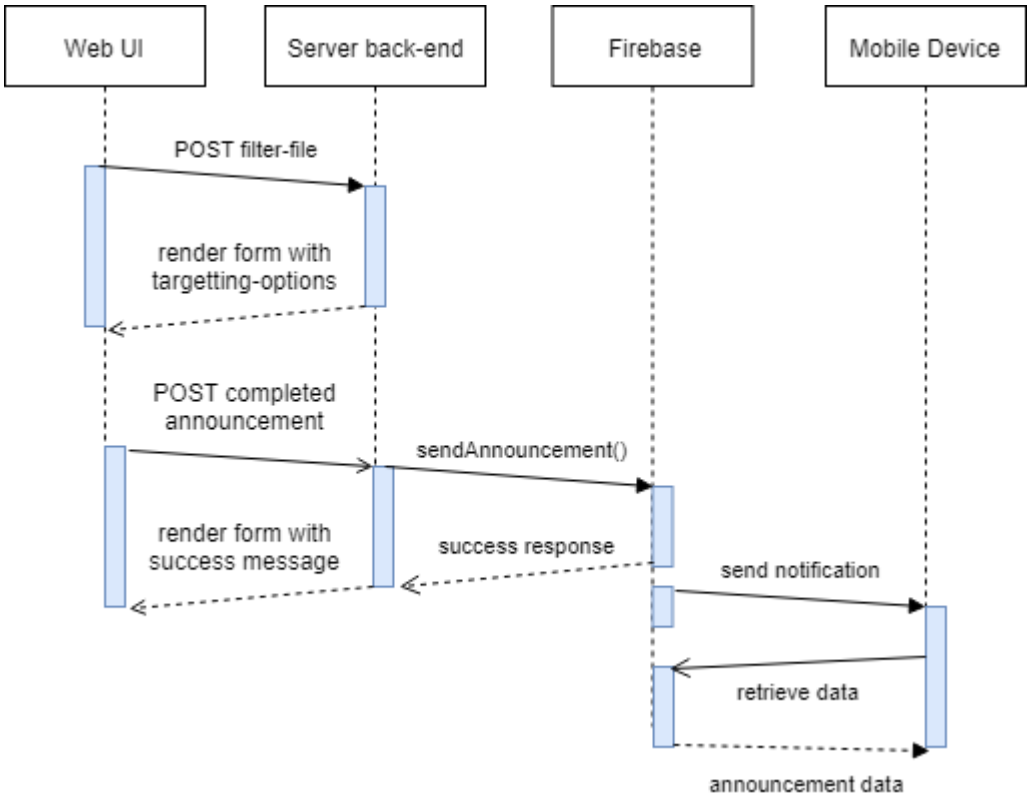
NOTE: Types of all attributes are String, unless otherwise stated, and in implementation, there may be extra attributes for a class.

Above is a class diagram illustrating our domain entities, and their relations. To clarify the elements of this diagram, there is an existence-dependency association from Event objects to Announcement objects. In other words, each event must be related to exactly one Announcement, while an Announcement object will optionally have one Event object associated to it. This means all events come with an announcement, but not all announcements come with an event.

The fact that there is no association line between Announcement, and Admin classes reflect the one-way reference between the two. This means that while an Announcement object has an author attribute referring to an existing Admin object, Admin objects have no information referring to any specific Announcement.

The above model is too simple, and the actual implementation too complex to efficiently show the design patterns we incorporated. Design pattern that we used most explicitly is Model View Controller (MVC) on all platforms, and this is reflected in the Sequence Diagram in next sub-section.

Interaction between components in MVC pattern



The Sequence Diagram above illustrates how the high-level components of SpiderByte interact to enable the use-cases of targeting a student group and sending an announcement.

We have the MVC pattern applied at various levels in the individual apps. At the highest level, each component illustrated above takes the corresponding role as listed below:

- Web UI – View
- Server back-end – Controller
- Firebase – Model
- Mobile Device – View

If necessary, read below for detailed explanation of the diagram.

After the *admin* is authorised, they get what we refer to as "filter-file" from Student Records to obtain a list of students to send an announcement to. Unlike the system-as-is, rather than copying the email column inside the filter-file into the emailing software, with SpiderByte, the admin simply uploads the file to a form-field. The server immediately responds with a version of the form that lists the different target-options from the actual filter-file.

Once all the required fields are filled, the admin then submits the announcement, which the server forwards to Firebase, and upon success response back, renders a success message.

Firebase then takes over to send notification of database change to the mobile devices. Mobile device will then request to retrieve the announcement data. When Firebase returns the announcement data, device will show the announcement depending on whether mailingList attribute of the announcement object contained the student email associated to instance of SpiderByte installed on the device.

Software quality

We tried to keep SpiderByte high quality in the following aspects by their associated actions:

- Reliability (comprehensively testing the modules, ensuring they are as independent)
- Adaptability (making changes as easy as possible)
- Persistence (making it hard to make mistakes, putting leading constraints, predicting errors, graceful and secure failing e.g. hiding server's error stack in production)

Our approach to testing and inspections:

We took testing, review, and inspection as important part of development, and included these in our definition of what a task "done" is. Therefore, a task is not "done" until tested, and reviewed. Following are some actions, and development techniques we employed in this aspect:

- Unit testing core functional modules
- Integration/component testing of alternative scenarios to check our constraints
- Code-review, programming collaborations (pair programming)
- Refactoring, documenting to make software useful beyond just our own development

Common challenges to software quality, and our solutions:

- There are complex procedures that are difficult to keep track of, and time-consuming to test. For example, for Web, browser testing of our constraints can take up to 10 minutes a session for a human. For this reason, we put effort in implementing automated tests although the process took significant amount of time itself (Find test mentioned here in Web/test subdirectory).
- Some functionalities had to be tested manually, including: notification sending, real-time database syncing on web, and mobile devices.
- Often, we rely on our libraries, and APIs to be well-tested. Although if those external functionalities are only a part of another functionality that we implement, we do test them.
- Different coding styles (we set up our project wiki early on for consistent programming practices)
- Team members are trained to different extents, so code quality can vary by member (we collaborate and review each other's code).
- Independent works can cause smelly code. For this reason, merging the independent works is often a work by itself. Here is where we utilise collaboration, and review once more.

Team organisation

Role management:

On the surface, our team followed the Agile Scrum software development process. We assigned the roles of Scrum master, product owner, and developer among the team.

Also, each team member belonged to one of the three platforms of SpiderByte, namely Web, iOS, and Android. Moreover, some team members took extra role in project management, and the deliverables. Following is attribution of the essential roles to each of the corresponding team member:

Munkhtulga Battogtokh	Web developer, Product owner, Scrum master
Adriel Aiach-Kohen	Web developer, Documenter
Henry Valeyre	Web developer, Documenter
Aakash Doshi	Android developer
Dimitris Papatheodoulou	Android developer, Documenter
Alin Fulga	Android developer, iOS developer
Mohammad Sadi	iOS developer, Product owner, Secretary
Maiwand Maidanwal	iOS developer
Kymani Lawrence	iOS developer, Documenter

Main communication lines:

General communication – Slack

Task-allocation, task categories – Trello (Kanban board)

Review/Inspection, pull requests – Github Enterprise

Decisions, project progress – Meetings, and Trello

Time management, in terms of working schedule:

Week 1-2 – Training based on initial project brief

Week 3 – Requirement elicitation, and analysis with client

Week 4 – Sprint 0 start (MUST tasks)

Week 5 – Sprint 1 start (SHOULD tasks)

Week 7 – Sprint 2 start (USEFUL tasks)

Week 9 – Sprint 3 start (USEFUL, and ADDITIONAL tasks)

Week 10 – Sprint 4 start (left-over tasks, and deliverables)

Task management:

Task completion – By our definition, a task is “done” only if it’s tested and reviewed. The member who is initially allocated the task is the one to write its test cases. Once they’ve done that, it’s also their responsibility to find a teammate who can review their code. At the most informal level, testing/reviewing were done through manual testing, and pull requests. Otherwise, we booked collaborative sessions with each other to merge, and clean the code together, and wrote automated tests.

Task prioritisation – Following the initial project brief, we prioritised the tasks with the highest business value, and highest effort first. Business value was measured by categorisation into one of the following (with decreasing values): MUST, SHOULD, USEFUL, and ADDITIONAL. We let the client eventually decide the priority of each task.

Conflicts – Client resolved any conflict regarding task priority whenever relevant, and otherwise we followed the majority.

Kaizen – “always improving” was our work-ethic philosophy for SpiderByte. We regularly held Sprint Reviews, and Retrospectives to keep track of progress, speed, and areas we could improve. As result, we took the following team-organisation actions *among others*.

- Task communication on Trello – for clarity and consistency, each task on Trello is required to at least have a label indicating priority, one indicating sub-team, and one indicating the assignee.
- Planned meetings – Every meeting was planned since Week 2. Hence, shorter meetings.
- Task detailed planning – For each tasks of the current sprint, one person creates a TODO comment detailing the task in the corresponding file where the task is to be implemented. Then each teammate is only told which TODO comments to find, and where. This made task allocation clear, and concrete so that it prevented members from not knowing what to do.