Design Document
Project for NordVisa
2DV603

Leif Karlsson

Johan Gudmundsson

Francis Menkes

Feiyu Xiong

Axel Nilsson

Table of Contents

Introduction

Purpose

This software design document describes the architecture and system design of an event calendar management system for the NordVisa organization. The system utilizes a client/server architecture and consists of two distinctive parts—the web based client software and the server backend system.

The client software is comprised of an event management dashboard and an embeddable web widget that connects to the backend API to present events entered by the system's users. The backend server exposes this API to the client and manages the connection to the database, while also handling user management and authorization. The database used is MongoDB, and its schema is described below.

The target audience for this design document is mainly the system's developers and future maintainers of the code base.

Requirements

The design document outlines the architecture of the requirements set forward in the software requirements specification. The event management (creation, viewing, updating, and deletion of events) and its pertinent API calls are outlined in accordance to requirements FR1-FR4.

The architecture responsible for user authorization and authentication (requirements F5-F13) is defined together with descriptions of its relevant software packages. Also, the software is designed using open source libraries and frameworks in accordance to requirement PR1 and ProcR-1. Requirement PR2 dictates that the client app should be responsive, something that is also taken into account in the design.

Design priorities

Major guiding priorities during the design process were ease of deployment and maintainability. The stakeholder's top priorities with regards to software requirements were also used as a starting point for the design, and acted as a catalyst from which further design decisions were made.

Glossary

Short	Long	Description
DAO	Data Access Object	An object there to abstract the communication between the application and the database

DTO	Data Transfer Object	In web application, we send data between the client and the server, but these data structures might only be used during the transition but not in the rest of the application. These are DTOs
DBMS	Database Management System	Software between the application and the raw data of the database which takes instructions from the application and then executes it on the raw data.
SPA	Single Page Application	A single-page application is a web application that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In a Single Page Application, either all necessary frontend code is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, in response to user actions.

General Priorities

In this design document, we have prioritised the two core functionalities, events handling and user management. We see these as the most important parts for the application. But we are still trying to fulfil as much of the requirements as we can.

To make the design a bit simpler to a beginning we have trimmed down some features like how we interpret and handle custom regions. We have also left out some of the less important features like automatic email on registration and password requests, and making a simple solution which can be replaces later with a real SMTP implementation.

Outline of the design

System

The system is built with an n-tiered architecture which consist of three layers, client, server, and, database. The client and the database cannot communicate without going through the server. And since we are using a separate client and server we are also using the client/server architecture. The three layers also uses an MVC-like architecture because client = view, server = controller, and database = model.

Decoupling and interchangeability between each layer has been a focus for us. The communication between client and server is mostly done through an HTTP API which is documented in this document. This would make it easy to create a new server application or client application while still using the old ones. The communication between server and

database is though the standard protocol of the database using a driver, so we tried to make that as easy as possible to change in the server application if you wish to use another database.

Client

The client is a JavaScript SPA using the library React. The application is divided into components where one component represents the UI and front-end logic for one specific view. The application request data from the server before a component mounts and sends input data from the user to the server.

Server

The server is a Java application running the Spring framework. This application both provides an API for the clients to use to make changes on the server, and also serve the clients to the internet. It consists of two large packages and a couple small ones. The larger packages User and Event are responsible for handling API requests which has to do with event management and user management. Then we have smaller packages like Image for handling uploading of images, Token package which generates and saves an API token to be used by the widget, DatabaseConnections contains classes delivering the database clients within the application, and at last the DatabaseServices which are services running in other threads that do regular services on the database.

Almost every package has a DAO to communicate the database, this is using the strategy pattern to make them interchangeable if you for example want to use another database. This ensures that you can modify this software to work with any database you want.

Database

We chose MongoDB as our database management system which is a document database since it had some very nice location based features which suited this application very well. The database will hold data about a user's account and all the events. It also holds some extra data used by the application such as temporary links, API tokens and so on.

In a document database, you can store objects nested within objects. We could make it so each User has an array of all events they have created, and this was the original idea. But we later realized that storing them separately makes more sense since mostly the events are not pulled from the database by who created the event, but by location and by time. So we ended up using the database a bit more like how you would use a relational database.

Major Design Issues

Overall system architecture

The NordVisa event calendar system is based on a so called n-layered, or multi-layered/multi-tier, client-server architecture—specifically a three-tier architecture where

presentation, domain logic and data persistence are separated in different layers. The client software is run in the web browser, while the domain logic and data persistence layers are run on a single backend server.

During the design process, different general architectures and sub-architectures were considered, for example a service-oriented architecture (SOA), a message-oriented or event-driven architecture, and a monolithic architecture. These alternatives are discussed below.

Service-oriented

While a service oriented architecture would improve the architecture in many ways we will opt not to use this type of architecture. Our reasoning is that deployment of this kind of architecture does become quite complex very quickly. This application will not be gigantic so we would not have a huge benefit from separating it into smaller services, but would make deployment more difficult.

Message oriented

We will not use a message oriented architecture since we don’t see a need to be able to update the clients in real time. We will instead use a RESTful API which requests data from the server in its current form. If the user wants to update their client they will have to request the data again. This will keep the architecture simpler.

Monolithic

A monolithic architecture would not work very well as a web application. The only possible way to do this would be to have the application only run in the browser, which would make the data created by one client inaccessible from any other client. A monolithic architecture would not work in this case.

Client/Server

The client/server architectural pattern is a natural fit for the NordVisa application. The centralized database requires some kind of server that will handle the connections to the web clients through an API. The web clients initiate a connection to the server to use its services.

MVC

Separating the UI from the data fits our application well. The view will be the client on the user's browser displaying the application, the controller is the http API the client uses to communicate with a server, the controller then tells the database which would be the models what to change or update.

N-layer

The n-layered architectural pattern works well for our application, with clearly defined and separated layers for presentation (web app), application (API), and data access layer (database).

Peer to Peer

This pattern would not fit very well with our problem because of our systems dependencies on persistent storage.

Server architecture

The NordVisa backend service will use a RESTful API that serves the web clients. It will run on the existing NordVisa virtual private server (VPS). For implementing the backend service, we have considered the following programming languages with accompanying web frameworks: Java, PHP, JavaScript and Python.

Java

Java has been chosen as the programming language for the NordVisa backend service. It's a mature and time-tested programming language with a multitude of production-grade web frameworks that have been used to implement very similar APIs as the one this software requires. While this is also true for many other programming languages, the fact that Java is one of the most popular languages will facilitate future maintenance and deployment. As described below, the Java based web framework Spring greatly simplifies deployment by packaging all required components in a single file—another factor that influenced our choice of programming language.

Java version

The currently installed version of Java on NordVisa's server is version 6. While making an application which would work on Java 6 would make deployment on NordVisa's server much easier, the final product would be out of date and insecure from the start. Java 6 reached its end of life in 2013 and has not received any security updates since. This makes Java 6 an insecure platform to use. Another problem this creates is that most tools and libraries no longer support Java 6 which would limit what we can use in the application.

We instead suggest updating from Java 6 to a newer version and with that update any software using Java since they are not secure running on old versions of Java. If some software still requires Java 6 then it is possible to run Java 6 in parallel with other versions of Java.

Because of the soon to be release of Java 9 which in turn will most likely lead to Java 7 soon reaching its end of life, we will go with Java 8 since that will most likely be supported for much longer.

Java frameworks

The Java landscape is home to a large selection of web frameworks. The ones we have considered are Spring/Spring Boot, Dropwizard and Java EE.

Spring/Spring Boot

Spring is a very popular open source Java framework that has matured into an enterprise standard in the last couple of years. Spring Boot is a pre-configured, opinionated framework that makes it easy to quickly get a Spring web app up and running. The ease of use and time-tested libraries combined with the ability to create a stand-alone, production-ready app with minimal configuration are some of the main reasons Spring Boot was chosen as the framework for the NordVisa backend service.

Dropwizard

Dropwizard is an open source web framework that has much in common with Spring Boot. There is an overlap in technologies and libraries used, and it's possible to create a stand-alone app that is easy to deploy. However, the ubiquity of Spring and its well-documented use made us choose it over Dropwizard, especially with maintenance in mind.

Java EE

Java Enterprise Edition (Java EE) and Spring/Spring Boot share much of the same core technologies, and using core Java EE APIs would surely present no obstacles in implementing the NordVisa backend. However, as mentioned above, Spring Boot's convenience and its ability to quickly get an application up and running is what ultimately led us to choose it as the preferred framework over Java EE and the others.

PHP

The current NordVisa website is built around WordPress and PHP. To keep things simple, it would have been attractive to leverage the existing architecture for the new event calendar. However, as one of the requirements is that the event calendar widget should be embeddable in any kind of website regardless of architecture, PHP was deemed unsuitable for this scenario.

JavaScript

Compared to PHP, there are no technical reasons why the API backend couldn't be implemented in JavaScript—and maybe more specifically, Node.js. The Node.js runtime coupled with a web framework like Express or Koa would no doubt suit this specific scenario. However, as we wanted to use existing technologies already present on the current NordVisa server, where Node.js is absent, we chose not to use JavaScript on the server side.

Python

Combined with mature web frameworks like Django or Flask, Python could have been used as the programming language of choice for the NordVisa backend service. Python is installed on the NordVisa VPS, which would have made deployment trivial. However, in order to leverage the current developers skillset as much as possible, Java was chosen over Python.

Database

The NordVisa database will use MongoDB, an open-source NoSQL document store database. The reasons for this choice, as well as other DBMS considered, are outlined below.

Requirements considered

The database system used must be open source in order to comply with PR-1. The database should have a built-in way to handle searching documents by location in order to minimize the time spent on fulfilling FR-14 and FR-16. FR-1 through to FR-12 are also relevant considerations, as these have to do with event CRUD and user authorization.

Entities required

We identified five different entities that need to be stored in the database based on the requirements. These entities are Organization, User, Event, and Token. Additionally, we decided to store the images in the database and this is a fourth entity that must be modelled.

Organization

The Organization entity holds an organization's name. An example of an Organization is the NordVisa organization.

Relationships:
Organization has many Users

User

The User entity stores a user's credentials, i.e. email and password. As the requirements state that no personal information should be stored in the database, only this minimal information is stored.

Relationships:
User has many Events

Event

The Event entity stores all the data related to an event, most importantly the event name and location in two different forms: coordinates and text, as well as any images attached to the event.

Relationships:
Event has one Image

Image

The Image event stores an image file in binary, along with relevant information such as its filename and file type.

Token

The Token entity stores API tokens for the website widget. It stores the number of requests, when the token is valid until, and if the token is still valid, among other things. The token may or may not be associated with a user account. This is because widget embedding code can be created without being logged in. The most important difference between a token created for a user and a token without a user are the maximum number of requests that that token can have.

Database management considered

MySQL

MySQL is widely-used and therefore well-tested and would suit the purposes of the NordVisa application. The community edition is open-source, but the enterprise edition is proprietary. But the lack of features for location handling discouraged us from using MySQL.

MariaDB

MariaDB was suggested by NordVisa as an alternative to MySQL at it is an open-source port of it. It is more or less the same as MySQL. However, This means it also has the same lack of features regarding location querying. This is why we did not choose MariaDB

Neo4j

We also looked at using a graph database for NordVisa, out of which Neo4j is one of the most popular. An advantage of using Neo4j is that it has built-in geospatial query functionality and would therefore suit our purposes. The major disadvantage is that none of us are familiar with graph databases or graph database query languages and this would incur a learning curve.

MongoDB

The benefits of MongoDB are that it is scalable and easy to design and adapt. The application's database is not very complex, only storing three different entities and few relationships between them. The most attractive feature of MongoDB, however, is the built-in location query functionality, where a query can easily be constructed for finding all documents within a specified range. The scalability, geospatial functionality, flexibility, and ease of design of MongoDB meant that it was the clear choice for the NordVisa calendar application.

Database Services

In our database we have data which should only exist for short temporary periods. We for example have temporary links which should only be usable for 24 hours, which will have to be stored in the database until they expire. And when they expire they should be removed from the database since they are of no use to anyone.

I saw the two following possible solutions

- A: Separate application running beside the main server application
- B: Functions running in separate threads

Solution	Coupling	Cohesion	Maintainability	Portability	Ease of use
A	Very Low	Very High	Very Good	Medium	Very Bad
B	Low	High	Very Good	Very Good	Very Good

When it comes to coupling and cohesion running the services in separate applications looks like a better idea since we are taking parts of the application which only has to communicate with the database and not the rest of the application.

But as discussed during the system architecture part, we will not go with a service oriented or microservice architecture because it will make deployment more complex making the software less portable and harder to use. This could be solved with good orchestration, but that's bringing in another kind of complexity which is not needed for a system this size.

Client

Summary

Other than the obvious that HTML, CSS and JavaScript is needed for the main layout and structure. Our main challenge was to be able to render the widget in a easy way on the websites where it is embedded. And easy for users to be able to embed the widget without

any technical or low-level knowledge. We came to the conclusion to use some kind of component based JavaScript library since it would allow us to dynamically render the widget as one single component, and mount it on the website with only one mounting points, which makes it easy to embed the widget on a website since the user only would need to link one external script. We also made the decision that it would be best to use the same set of techniques for the entire application (Widget and Dashboard). And since there are no downside of using a component based JavaScript library on for the dashboard as well, we had no reason to not use the same techniques for both.

Libraries & Frameworks considered

Following the above, we then took a look at the three most used component based JavaScript libraries **Angular2**, **React** and **Vue** since it would make future maintenance easier if the project would be continued in the future by other developers, it would be more likely that they have experience with these libraries compared to others. In the end we decided to use React. The reasons for this choice, as well as other libraries considered, are outlined below.

Angular2

Angular2 is widely-used and would suit the purposes of the NordVisa application. But we quickly came to the conclusion that Angular2 would not be a good fit for this project because of its full flagged framework nature.

Vue

Vue would suit the purposes of the NordVisa application. But it is a fairly new library and no one in our team has any experience with it. Since it is new, it could also be hard to find other developers with experience with it if any future development is needed.

React

React is widely-used and would suit the purposes of the NordVisa application. We also have members in our team with previous experience with React which at the end was the main reason to why we chose to use React for this project.

List of Libraries/Dependencies

Summary

Besides React, ReactDOM is also needed for the DOM manipulation and rendering for both the main application and the widget. And so is Babel to transpile the JSX code to vanilla JavaScript code.. For the main site also uses the following dependencies React-router to handle URL routes, Prop-types to be used with React context object for globally accessing language files, Moment to format the epoch date/time formats sent from the server to a readable format, Validator to validate email addresses and the components React Gcaptcha and Google Maps React which is used for implementing reCaptcha and Google Maps in an easy way. Google Map React is also used for the widget.

Main site/Dashboard

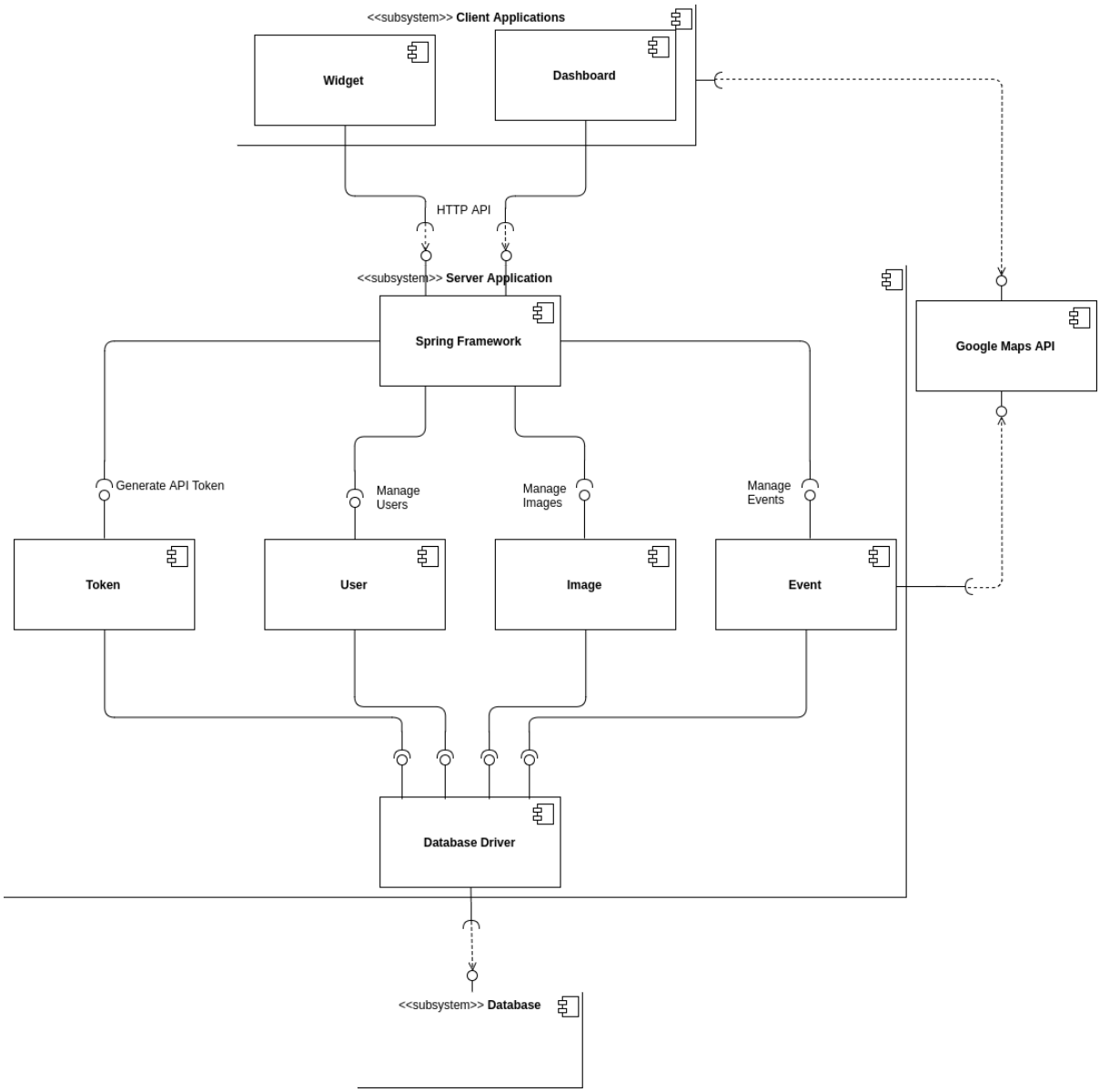
- React
- ReactDOM
- React-router
- Babel
- Google Map React
- Moment
- Prop-types
- React Gcaptcha
- Validator
- Babel

Widget

- React
- ReactDOM
- Google Map React
- Babel

Design Details - Software Architecture

Component Diagram



Architectural Patterns

Server-Client

When working with web application the Server-Client pattern is quite natural to use. Especially if you are making an application with persistent data you do need both this pattern to stay both decentralized and centralized at the same time.

Our Client and Server communicates with an HTTP API using the methods supported by modern browsers.

N-Layered

The n-layered or n-tiered architecture is quite common and work very well on application which only needs a few layers, but it can also grow too large and a large part of the code ends up only transferring data to another layer. But since we only need three layer this was perfect for us.

MVC

The MVC architecture was never planned, happened to fit what we had created with the other two patterns. We have three layer, database = model, browser client = view, and server application = controller.

Component Interfaces

Client - Server API

Event Requests

Name	Method	URL	Data

Get Event	GET	/api/event/get?id=""	Res [{
			id: "",
			name: "",
			location: {
			coordinates: {
			type: Point
			coordinates: [
			0.0,
			0.0
			]
			},
			address: "",
			parsedAddress: "",
			postalCode: "",
			city: "",
			county: "",
			country: ""
			},
			description: "",
			startDateTime: "",
			duration: 0,
			recursive: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdAt: "",
			updatedAt: "",
			createdBy: "",
			editedBy: ""

			}]

Get Events	GET	/api/event/get?longitude=””&latitude=””&radius=""&country=""&county=""&fromDate=""&toDate=""&token=””	Res [{
			id: "",
			name: "",
			location: {
			coordinates: {
			type: Point
			coordinates: [
			0.0,
			0.0
			]
			},
			address: "",
			parsedAddress: "",
			postalCode: "",
			city: "",
			county: "",
			country: ""
			},
			description: "",
			startDateTime: "",
			duration: 0,
			recursive: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdAt: "",
			updatedAt: "",
			createdBy: "",
			editedBy: ""
			}]

Get Manageable Events	GET	/api/event/get_manageable	Res [{
			id: "",
			name: "",
			location: {
			coordinates: {
			type: Point
			coordinates: [
			0.0,
			0.0
			]
			},
			address: "",
			parsedAddress: "",
			postalCode: "",
			city: "",
			county: "",
			country: ""
			},
			description: "",
			startDateTime: "",
			duration: 0,
			recursive: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdAt: "",
			updatedAt: "",
			createdBy: "",
			editedBy: ""
			}]

Get All Events	GET	/api/event/get_all	Res [{

			id: "",
			name: "",
			location: {
			coordinates: {
			type: Point
			coordinates: [
			0.0,
			0.0
			]
			},
			address: "",
			parsedAddress: "",
			postalCode: "",
			city: "",
			county: "",
			country: ""
			},
			description: "",
			startDateTime: "",
			duration: 0,
			recursive: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdAt: "",
			updatedAt: "",
			createdBy: "",
			editedBy: ""
			}]

Create Event	POST	/api/event/create	Req {
			name: "",
			location: "",
			description: "",
			startDateTime: "",
			duration: 0,
			recurring: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdBy: ""
			}

Update Event	POST	/api/event/update	Req {
			name: "",
			location: "",
			description: "",
			startDateTime: "",
			duration: 0,
			recurring: boolean,
			recursEvery: "",
			recursUntil: "",
			images: "",
			createdBy: "",
			editedBy: ""
			}

Delete Event	POST	/api/event/delete	Req {
			id: "",
			}

User Requests

Name	Method	URL	Data

Register User	POST	/api/visitor/registration	Req {
			email: "",
			password: "",
			passwordConfirmation: "",
			organization: "",
			gRecaptchaResponse: “”
			}
			
			Res {
			success: boolean,
			errorCodes: [“”,””]
			}

Unregister User	POST	/api/user/unregister	Req {
			id: ""
			}

Update User Details	POST	/api/user/update_user_details	Req {
			id: “”,
			email: "",
			organization: ""
			}

Change password	POST	/api/user/change_password	Req {
			id: “”,
			oldPassword: "",
			password: "",
			passwordConfirmation: ""
			}

Get User By ID	GET	/api/user?id=””	Res {
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			}

Get User By Email	GET	/api/user?email=””	Res {
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "”,
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			}

Get All Users Within Organization	GET	/api/user?organization=””	Res [{ id: "", email: "",

			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			},
			...
			]

Get All Users	GET	/api/user	Res [
			{
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			},
			...
			]

Request password recovery	POST	/api/visitor/request_password_recovery	Req {
			email: ""
			}

Set password from password recovery	POST	/api/visitor/recover_password	Req {
			urlId: "",
			password: "",
			passwordConfirmation: ""
			}

Verify Email Address	GET	/api/visitor/verify_email?id=""	Res {} Redirect "/"

Get Pending Registrations	GET	/api/admin/registrations	Res [
			{
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			},
			...
			]

Accept / Deny Registration	POST	/api/admin/registrations	Req { id: "",

			approved: boolean }

Make User	POST	/api/admin/make_user	Req {
			id: ""
			}

Make Administrator	POST	/api/admin/make_admin	Req {
			id: ""
			}

Make Super Administrator	POST	/api/super_admin/make_super_admin	Req {
			id: ""
			}

Get current user	GET	/api/user/current	Res {
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			}

Get all organizations	GET	/api/visitor/organizations	Res ["", "", ...]

Get manageable members for current user	GET	/api/admin/manageableUsers	Res [
			{
			id: "",
			email: "",
			role: "",
			createdAt: "",
			updatedAt: "",
			organization: {
			name: "",
			approved: boolean,
			changePedning: ""
			}
			},
			...
			]

Other Requests

Login	POST	/login	Req {
			username: "",
			password: ""
			}

Log Out	POST	/logout	-
Generate Widget	GET	/api/token	`Res { token: "" }`

Upload Image	POST	/api/upload	`Req { files: [file] }`
			
			`Res { success: boolean, path: "" }`

Get Image	GET	/api/upload/{path}/{name}	`Res { [Image] }`

The response when there is an error will have the following format:

```
Res {
    timestamp: “”,tech
    status: xxx,
    error: “”,
    exception: “”,
    message: “”,
    path: “”
}
```

Server - Database

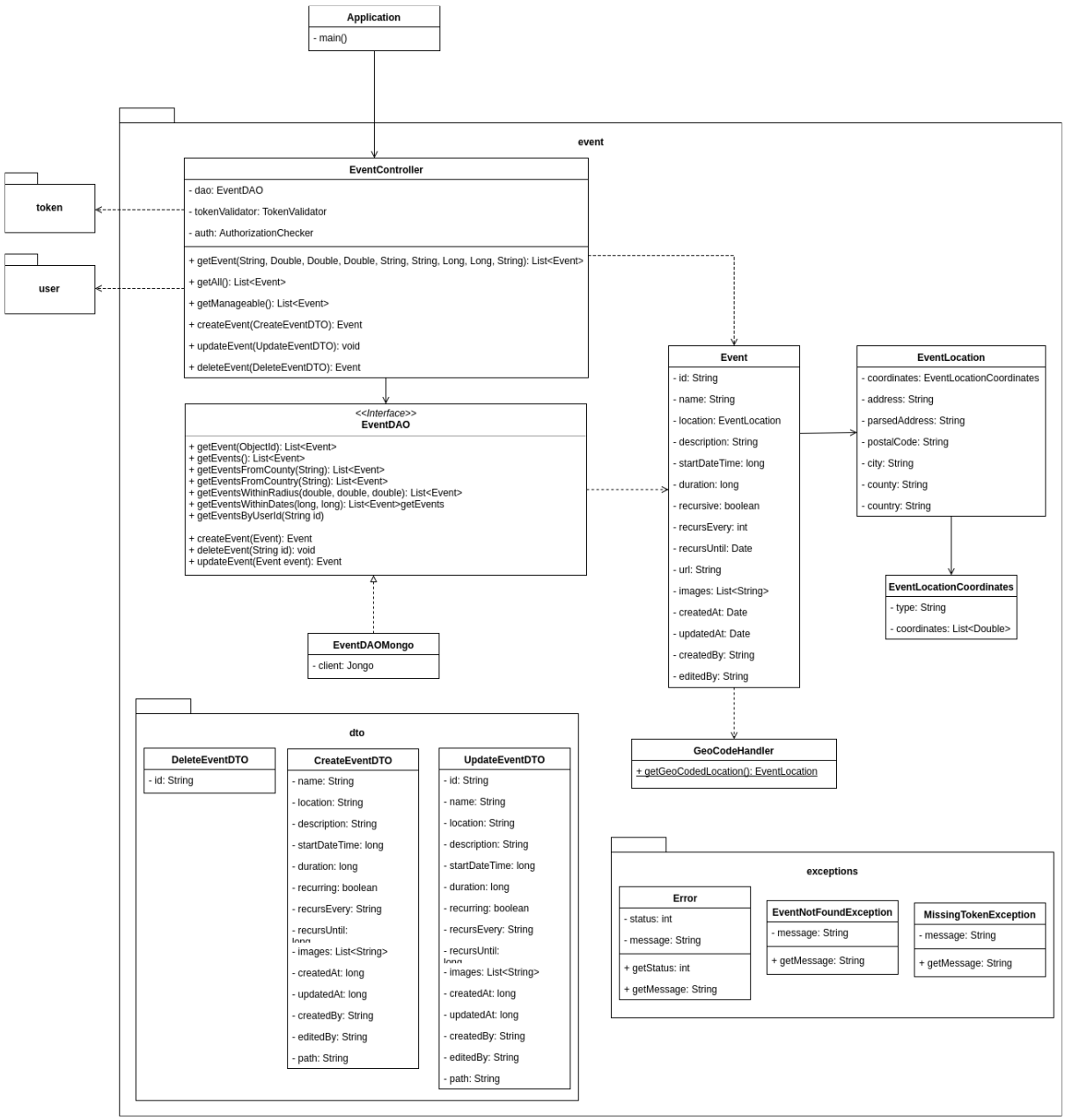
The communication between the server and the database is done using the protocol of that particular database. We have chosen to use MongoDB, so we will be using a driver for Java which will communicate with the database in the mongodb protocol.

We have also gone to great efforts to make this interchangeable with other databases. So implementing a different database should be straightforward.

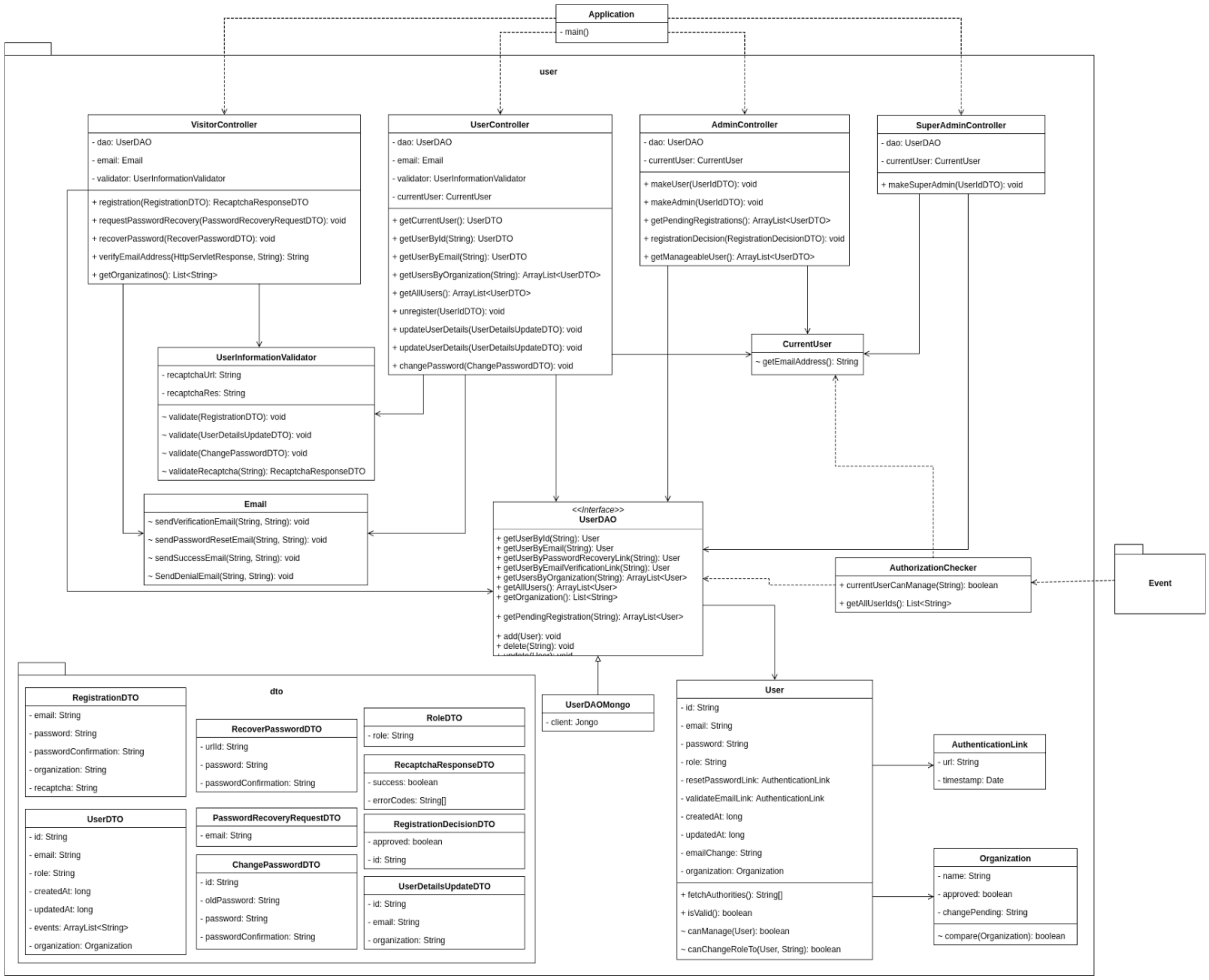
Design Details - Component Implementation

Class Diagrams

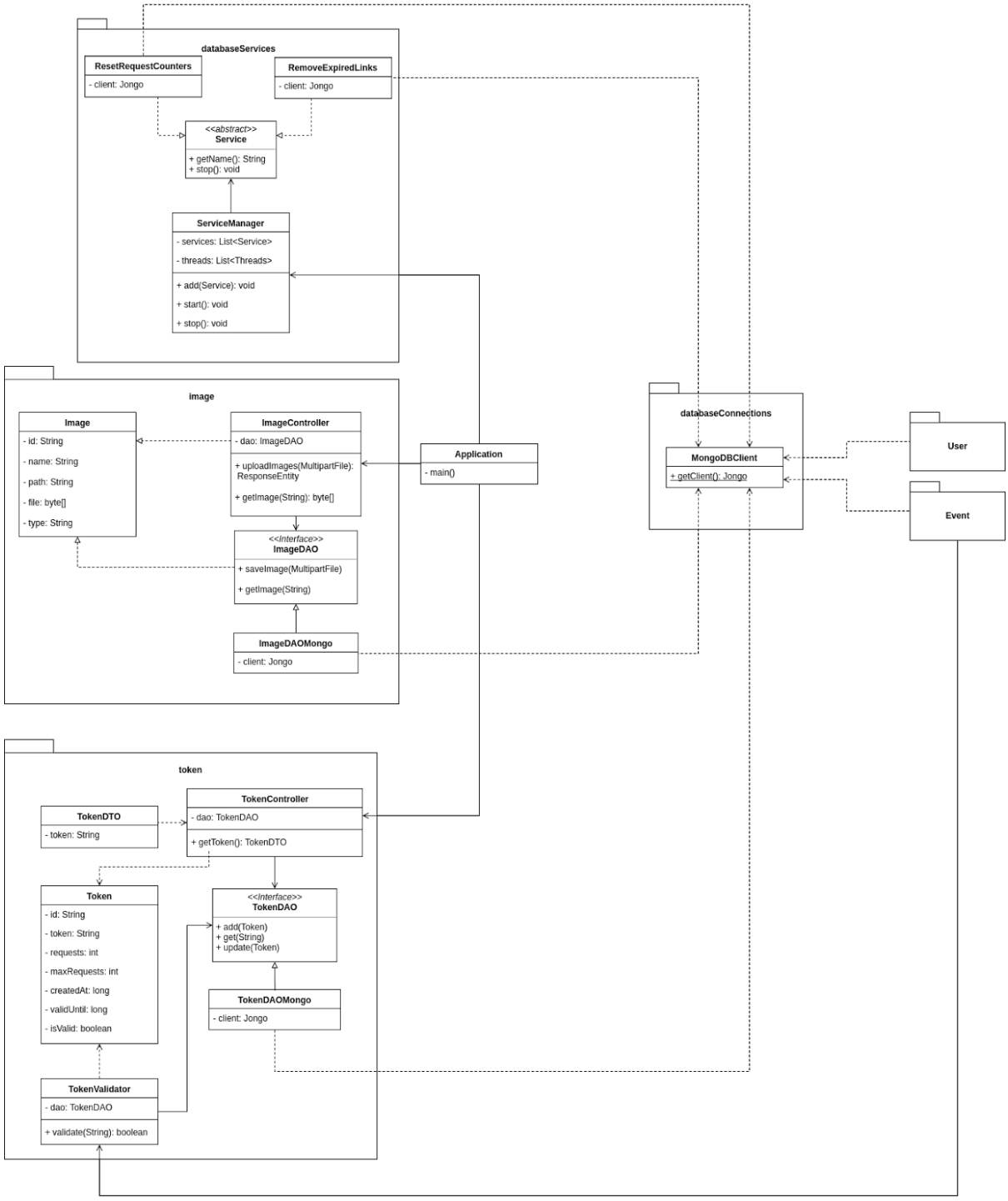
Event Package



User Package



Other Packages



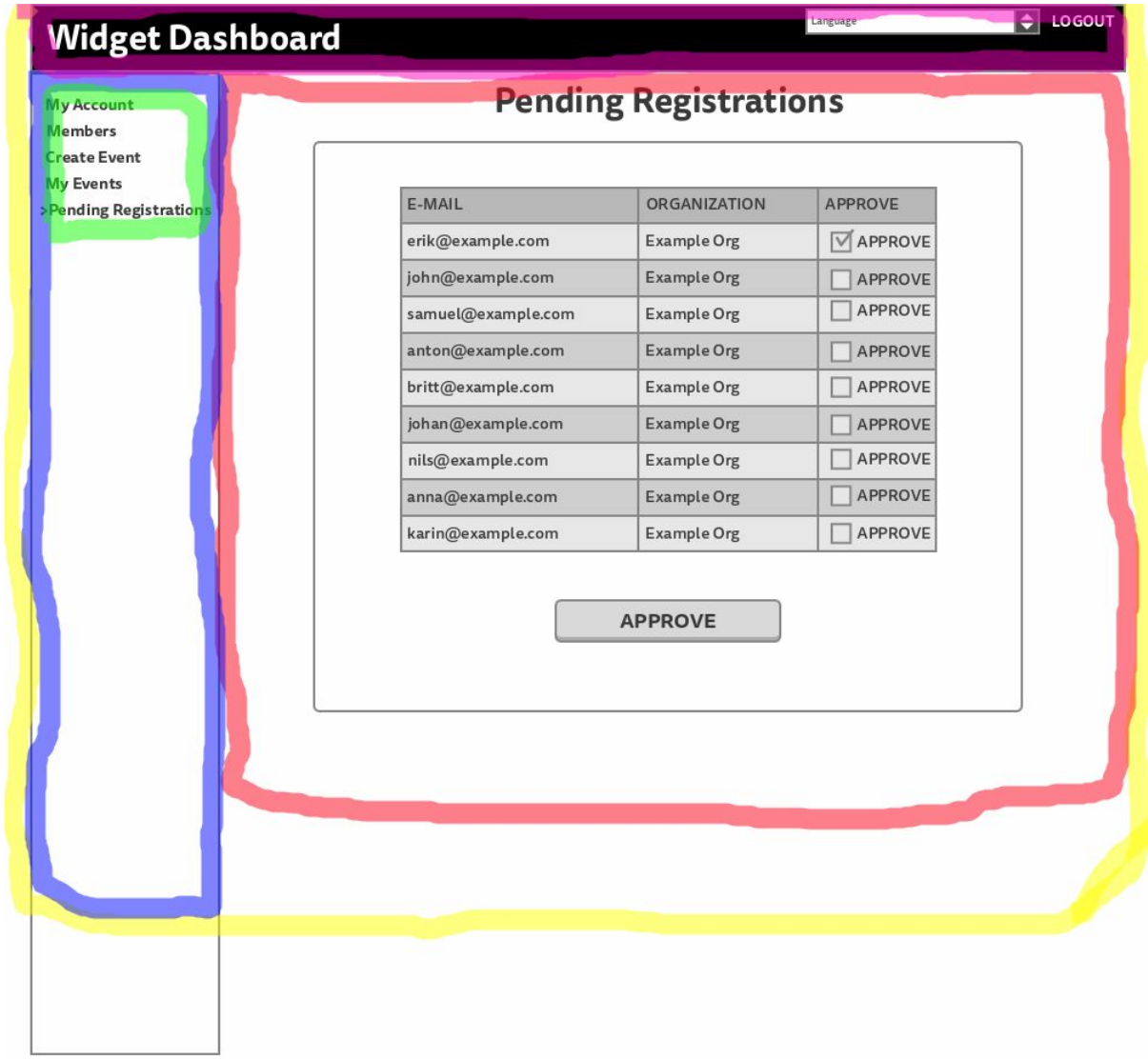
Client

Routes

Name	URL	Description
Login	/login	The login page
Logout	/logout	Performs logout and redirects to /login
Register	/register	The page for register a new account.
Recover lost password	/recover_password/	The page for requesting a password recovery email
Recover Password	/update-password/:url_id	The change password page to change your password without knowing your old one. The url_id is only known to the email owner and the system
Generate Widget	/generate-widget	The page for generating the widget code to be embedded into a website.
My Account	/user/account	The page for viewing account details and/or changing organization, email and password.
Create Event	/user/event/create	The page for creating a new event.
My Events	/user/event	The page for viewing all the event the user can manage
Edit Event	/user/event/edit/:id	The page for editing an already existing event.
View Event	/user/event/view/:id	The page for viewing an event
Members	/admin/members	The page for viewing all registered members
Pending Registrations	/admin/pending-registrations	The page for viewing all pending registrations

Identifying Components & Component States

Application



Yellow: App Component. Has all other components as child-components and is the component who is mounted to the HTML.

Purple: TopBar Component. Renders the applications name and is responsible for language selection for the entire application.

Red: View Container. Is the component who keeps track of the routes and mounts the correct View-components when they are asked for in the routes.

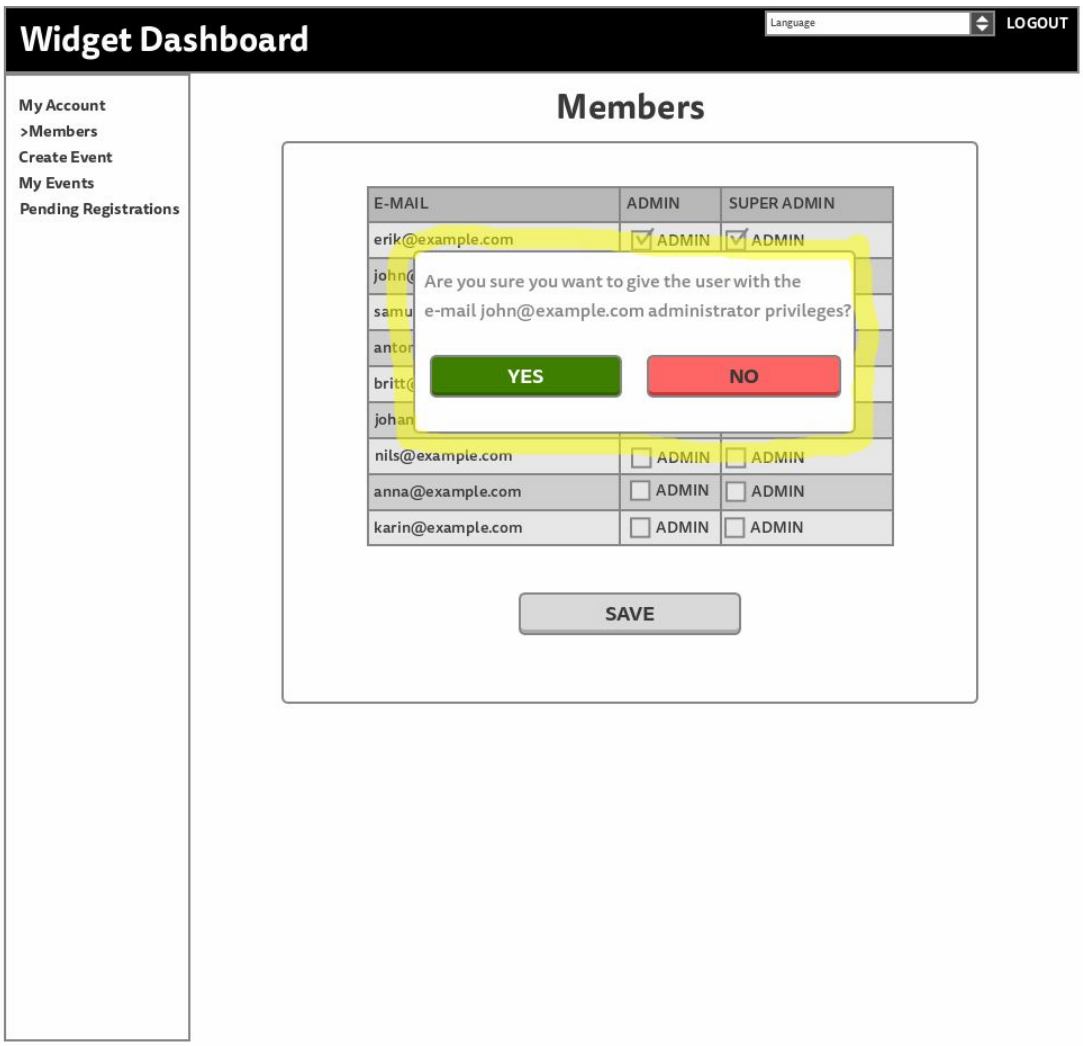
Blue: ResponsiveMenu Component is the container component for the menu and makes sure that it is responsive and fits every device. Has MenuList as a child-component

Green: MenuList component. Is responsible for rendering the Menu items and makes sure so the correct menu items are rendered based on if the user is logged in or not and if they are an administrator or not.

States:

Languages: Object - An object which keep tracks of all the language files
currentLanguage: String - Stores the current language selected, which is sent to context to be accessed in the entire application.

ConfirmMessage

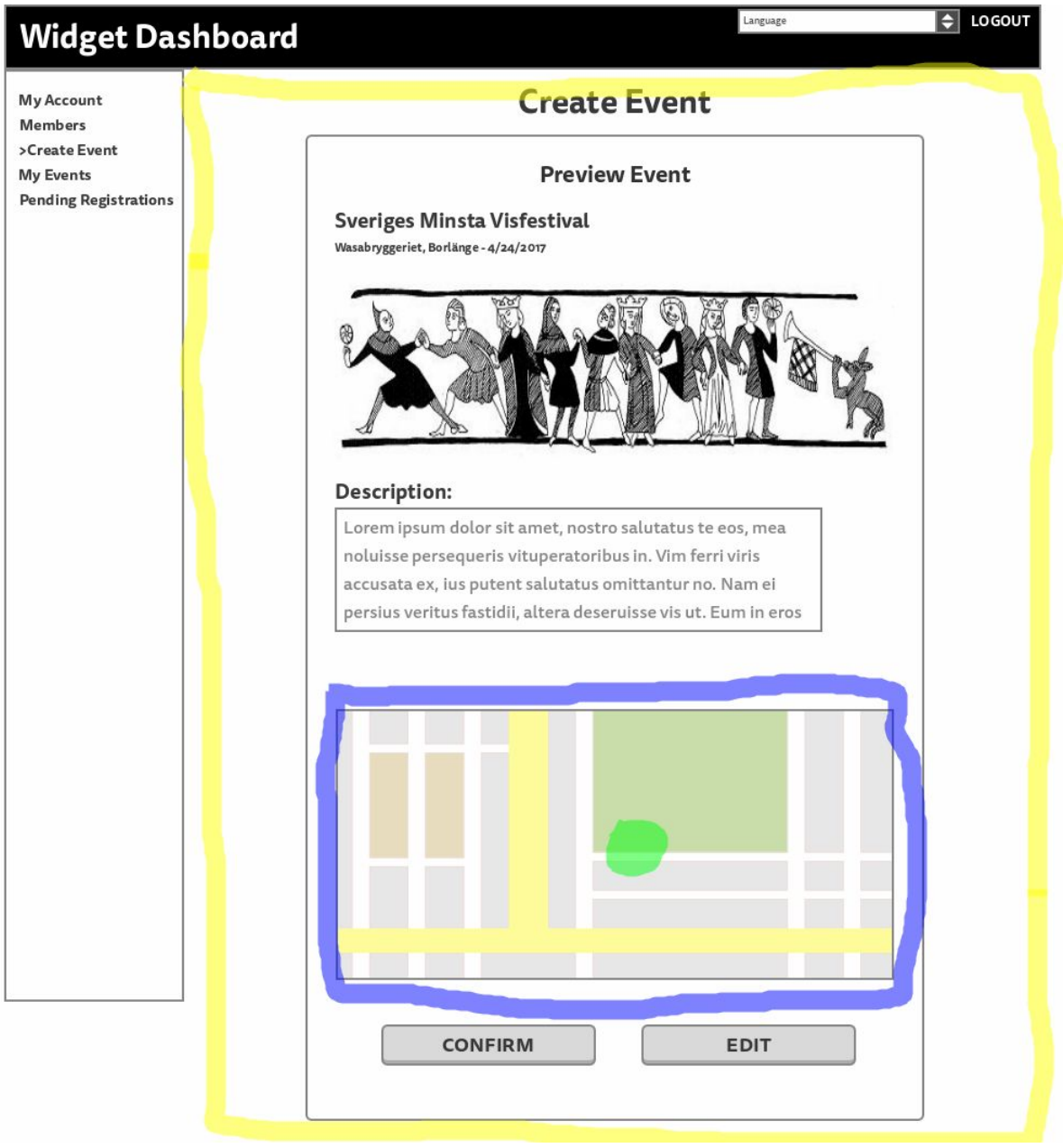


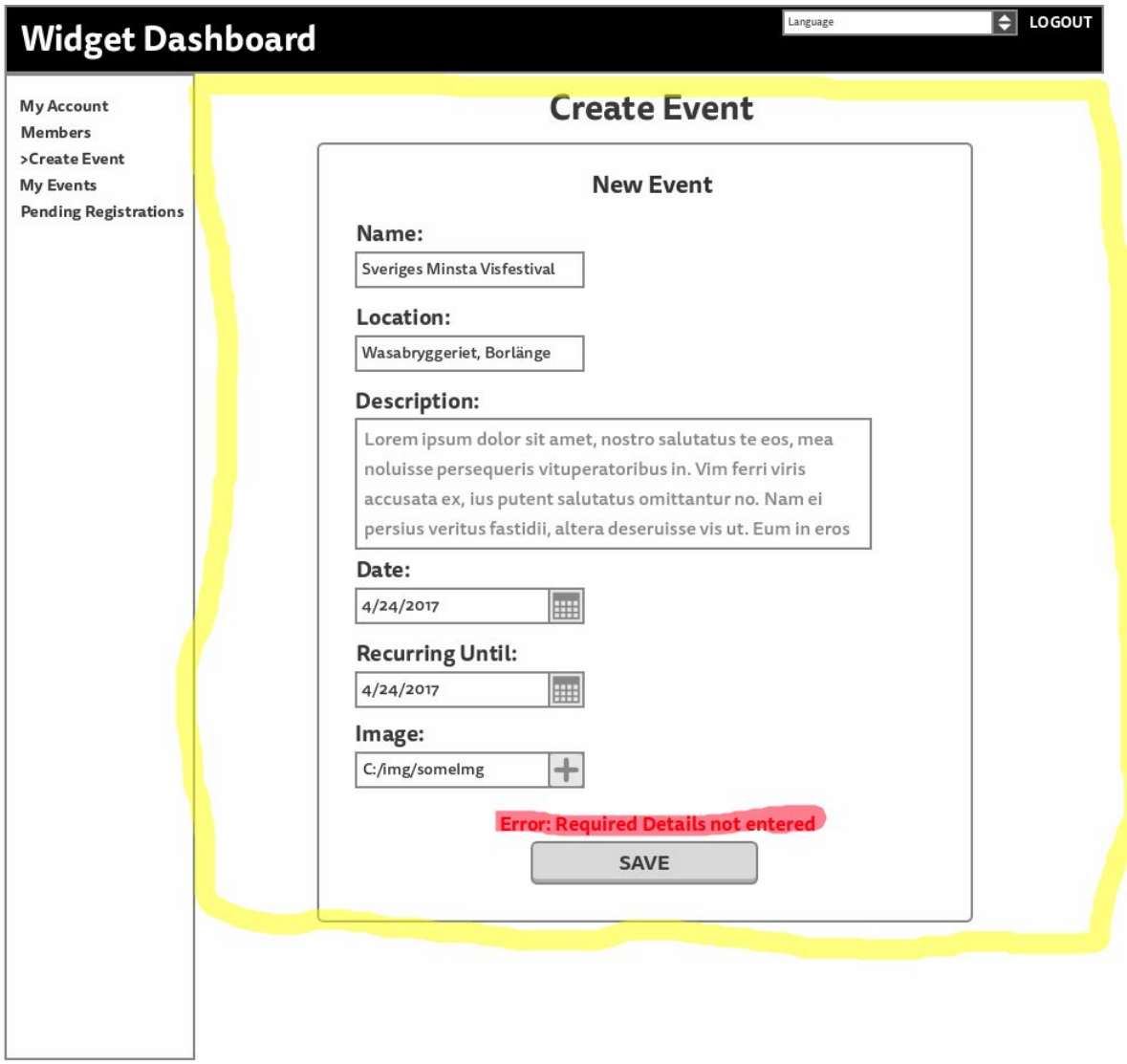
Yellow: ConfirmMessage Component. Is the component responsible for rendering a Message and informing the parent component when the user clicks yes so that the parent component can perform appropriate actions.

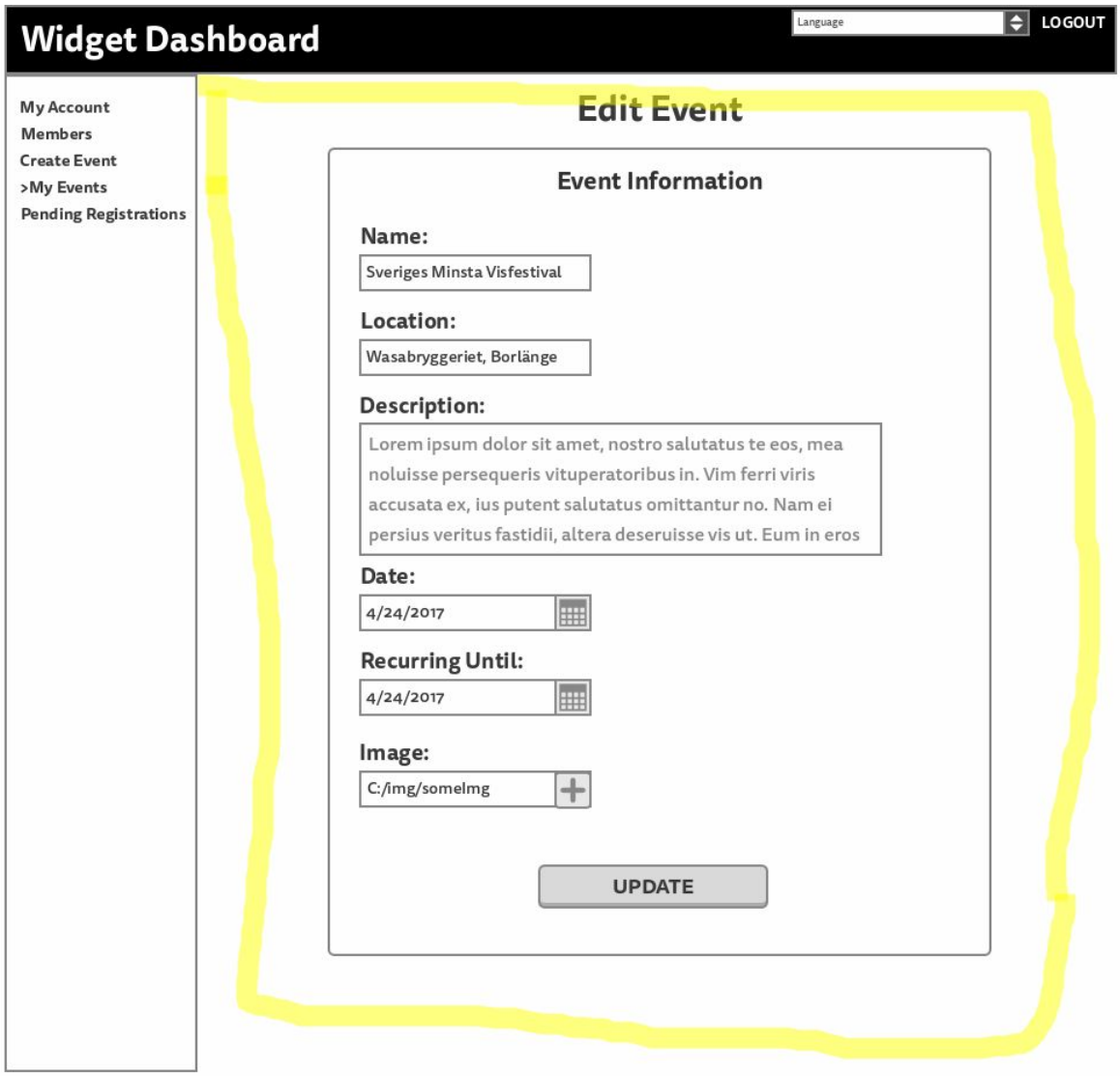
States:

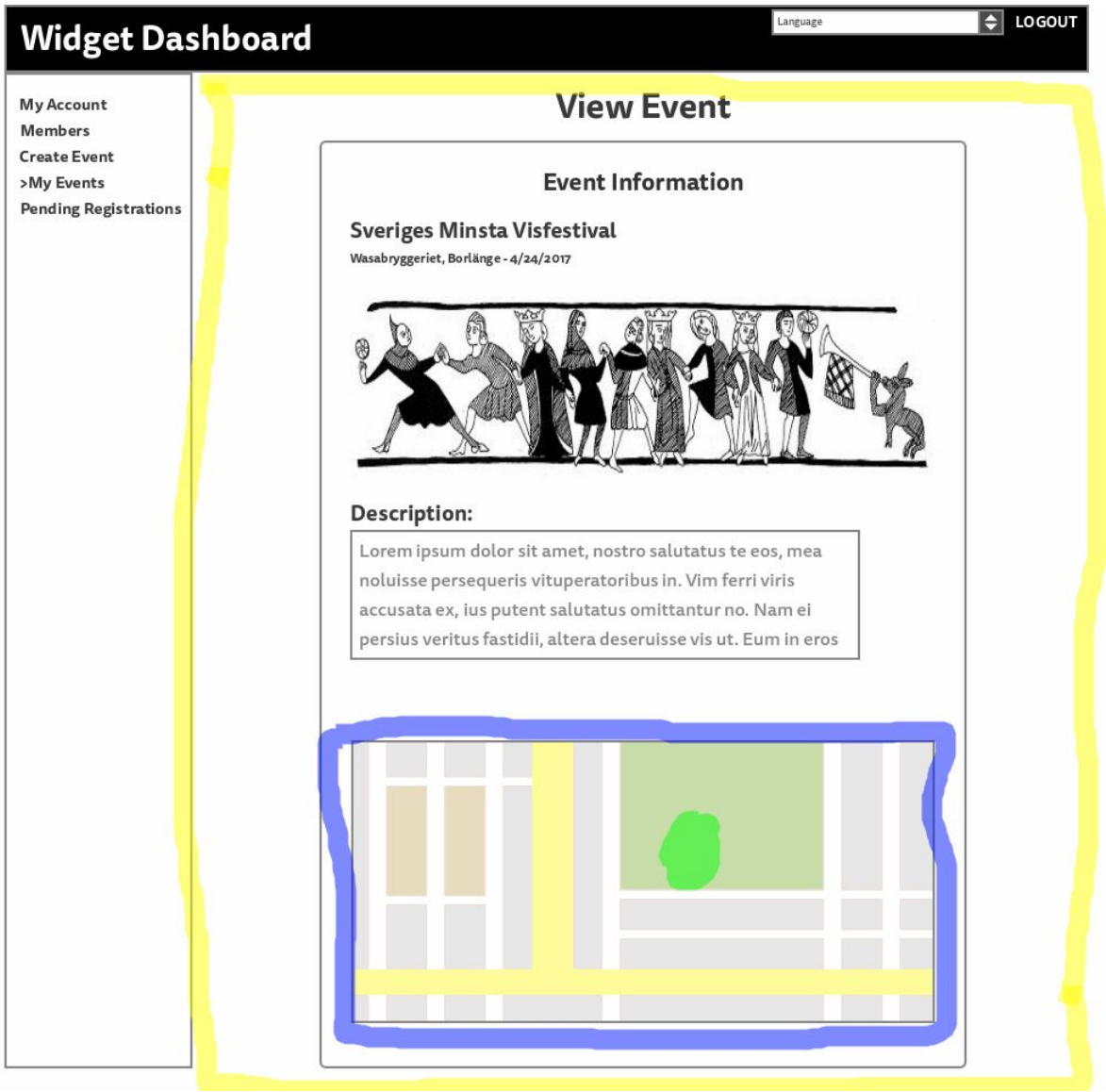
Pop: Boolean - Keep track if the message should popup or not.

CreateView


Widget Dashboard
Language
LOGOUT
My Account
Members
>Create Event
My Events
Pending Registrations
Create Event
Preview Event
Sveriges Minsta Visfestival
Wasabryggeriet, Borlänge - 4/24/2017
Description:
Lorem ipsum dolor sit amet, nostro salutatus te eos, mea noluisse persequeris vituperatoribus in. Vim ferri viris accusata ex, ius putent salutatus omittantur no. Nam ei persius veritus fastidii, altera deseruisse vis ut. Eum in eros
CONFIRM
EDIT


Widget Dashboard
Language
LOGOUT
My Account
Members
>Create Event
My Events
Pending Registrations
Create Event
New Event
Name:
Sveriges Minsta Visfestival
Location:
Wasabryggeriet, Borlänge
Description:
Lorem ipsum dolor sit amet, nostro salutatus te eos, mea noluisse persequeris vituperatoribus in. Vim ferri viris accusata ex, ius putent salutatus omittantur no. Nam ei persius veritus fastidii, altera deseruisse vis ut. Eum in eros
Date:
4/24/2017
Recurring Until:
4/24/2017
Image:
C:/img/someImg
Error: Required Details not entered
SAVE


Widget Dashboard
Language
LOGOUT
My Account
Members
Create Event
>My Events
Pending Registrations
Edit Event
Event Information
Name:
Sveriges Minsta Visfestival
Location:
Wasabryggeriet, Borlänge
Description:
Lorem ipsum dolor sit amet, nostro salutatus te eos, mea noluisse persequeris vituperatoribus in. Vim ferri viris accusata ex, ius putent salutatus omittantur no. Nam ei persius veritus fastidii, altera deseruisse vis ut. Eum in eros
Date:
4/24/2017
Recurring Until:
4/24/2017
Image:
C:/img/someImg
UPDATE



Yellow: CreateView Component responsible for rendering the correct UI depending on if the user want to view, create, preview or edit an event.

Blue: EventsMap Component. Responsible for rendering a map at the correct location and loop out the correct amount of MapMarker Components.

Green: MapMarker Component. Responsible for rendering a marker on the map at the correct longitude and latitude.

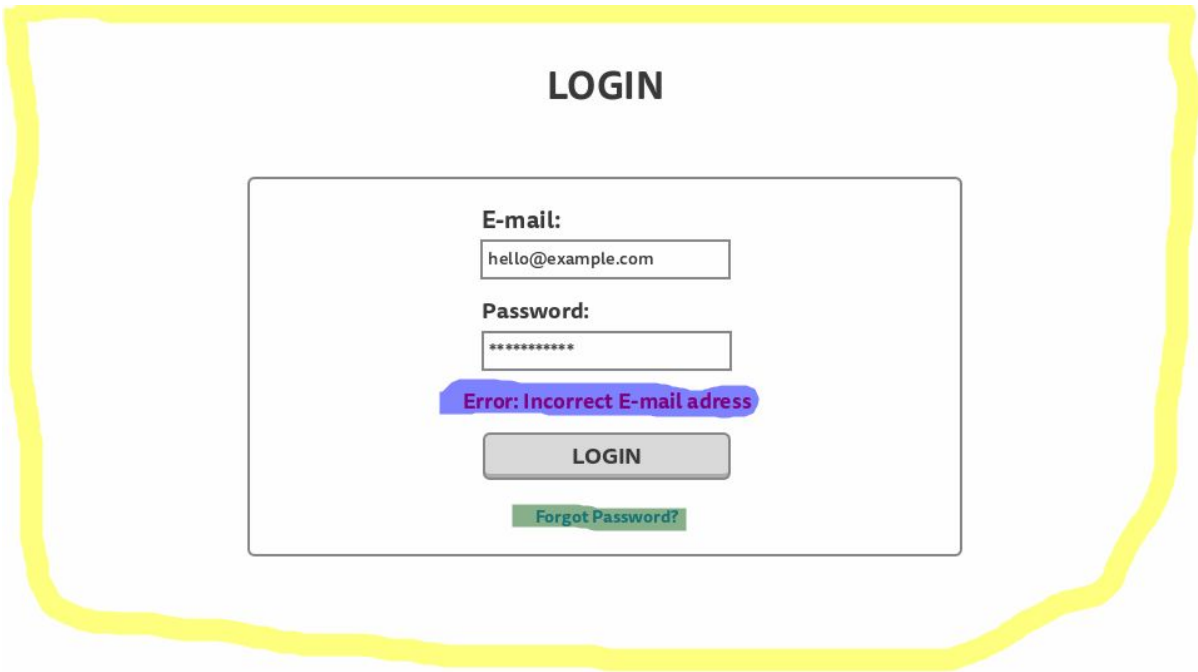
Red: ErrorList Component. Responsible for rendering error-messages if there are any.

States:

fields: Object - Which keep track of all the input data from the user.
fieldErrors: Array - Stores all the error messages which should be rendered.
progress: String - Keep tracks of where in the create event process the user is (create/view/preview/edit)

event: Object - The event requested from the server when the event is created.
comeFrom: String - The page the user was on before he entered the create event page.

LoginView



Yellow: LoginView Component Responsible for rendering the LoginView

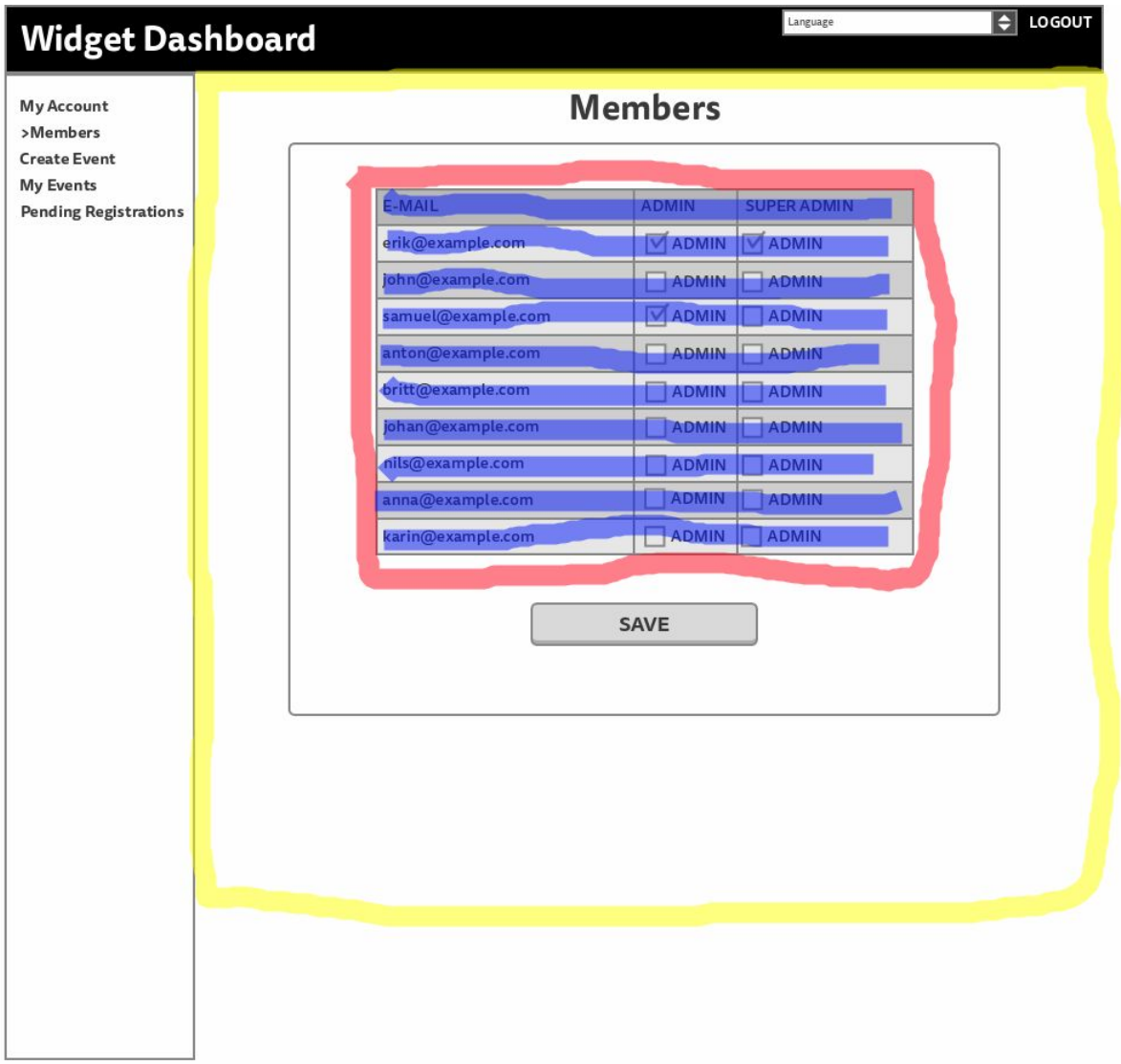
Blue: ErrorList Component. Responsible for rendering error messages if there are any.

Green: React Router Link Component. Responsible for updating the route to /recover-password

States:

fields: Object - Keep track of all the input data from the user
fieldErrors: Array - Store all the error messages that should be rendered.
loginInProgress: Boolean - Keeps track on if a loading symbol should be visible while waiting on a response from the login.
shouldRedirect: Boolean - Keeps track on if the user is logged in and should be redirected to My Events Page.

MembersView



Yellow: MembersView Component Responsible for rendering the Members Page View.

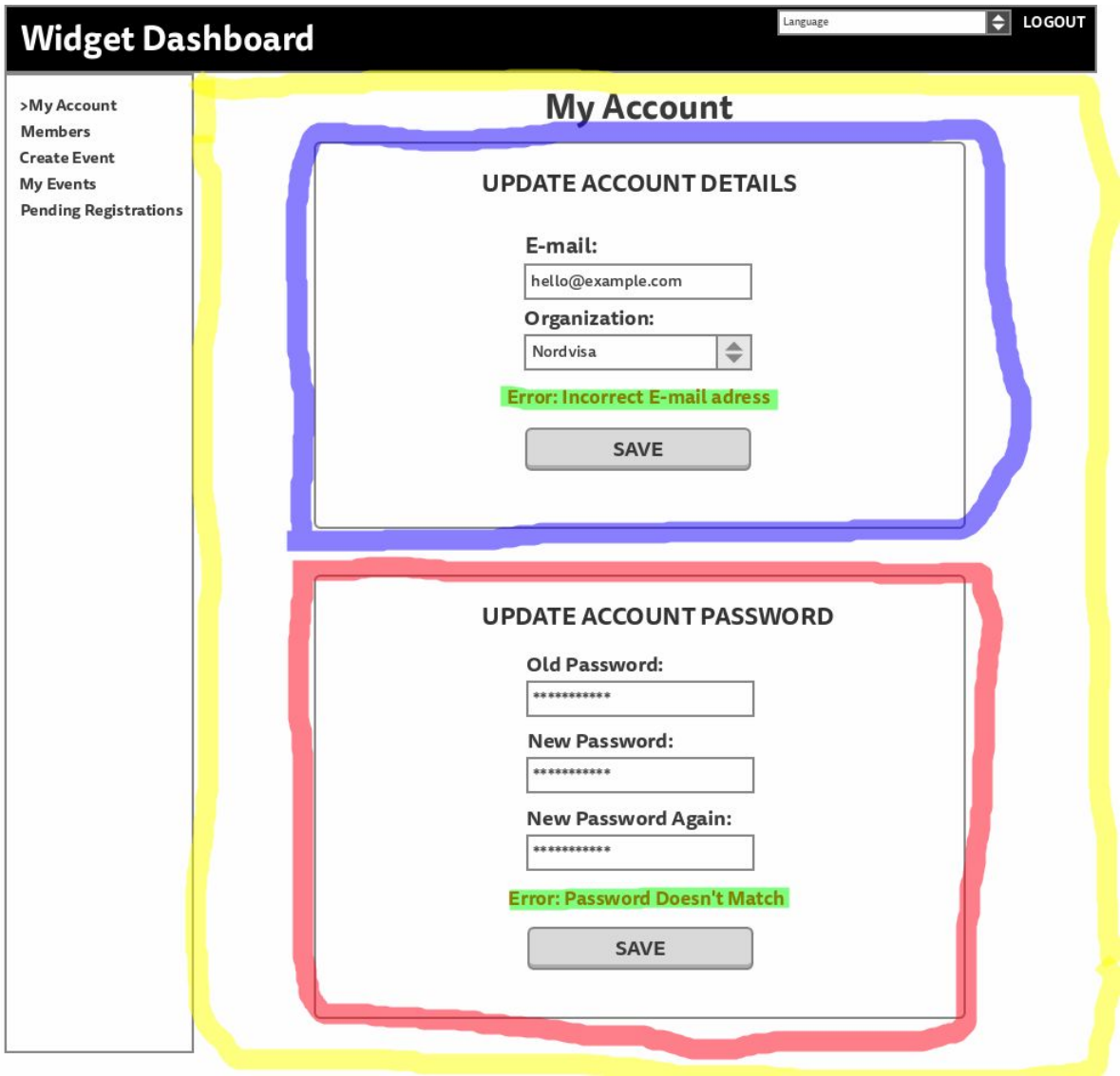
Red: MembersList Component responsible for rendering the list-header and loop out the right amount of Member Components.

Blue: Member Component. Responsible for rendering a member item in the list.

States:

members: Array - Stores all the members requested from the server
updated: Array - Keeps track of all the action taken by the user before submit.
popup: Object - Store the configuration for the popup which will be sent to ConfirmMessage Component.

MyAccountView:



Yellow: MyAccountView Component responsible to render the My Account Page with the child-component's Update Account and Update Password.

Blue: Update Account Component. Responsible for rendering the form for updating organization and email.

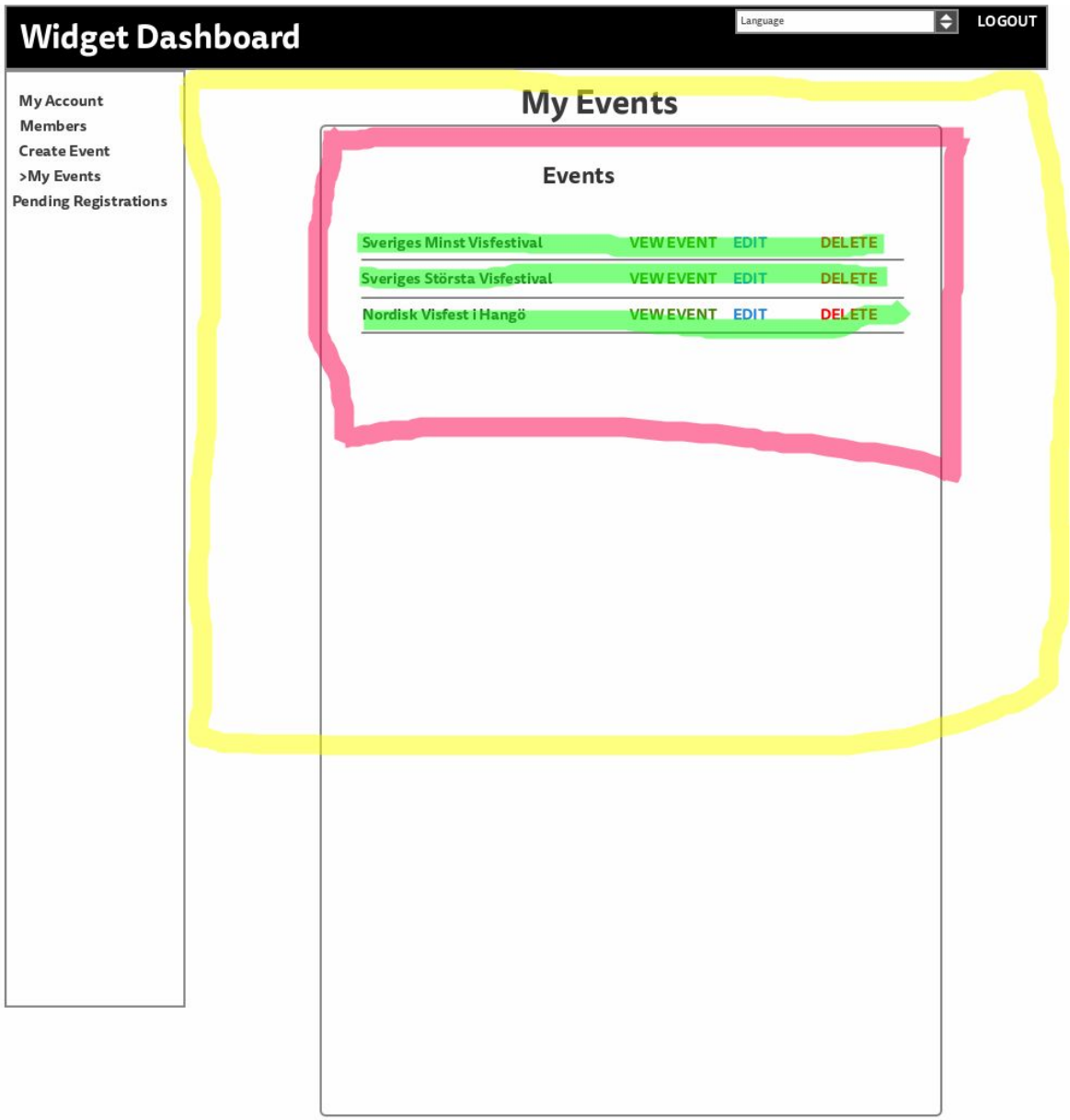
Green: ErrorList Component. Responsible for rendering error messages if there are any.

Red: UpdatePassword Component. Responsible for rendering the form for changing password.

States:

None

MyEventsView



Yellow: MyEventsView Component. Responsible for rendering the My Events Page.

Red: EventsList Component. Responsible for rendering the list header and loop out the right amount of Event Components.

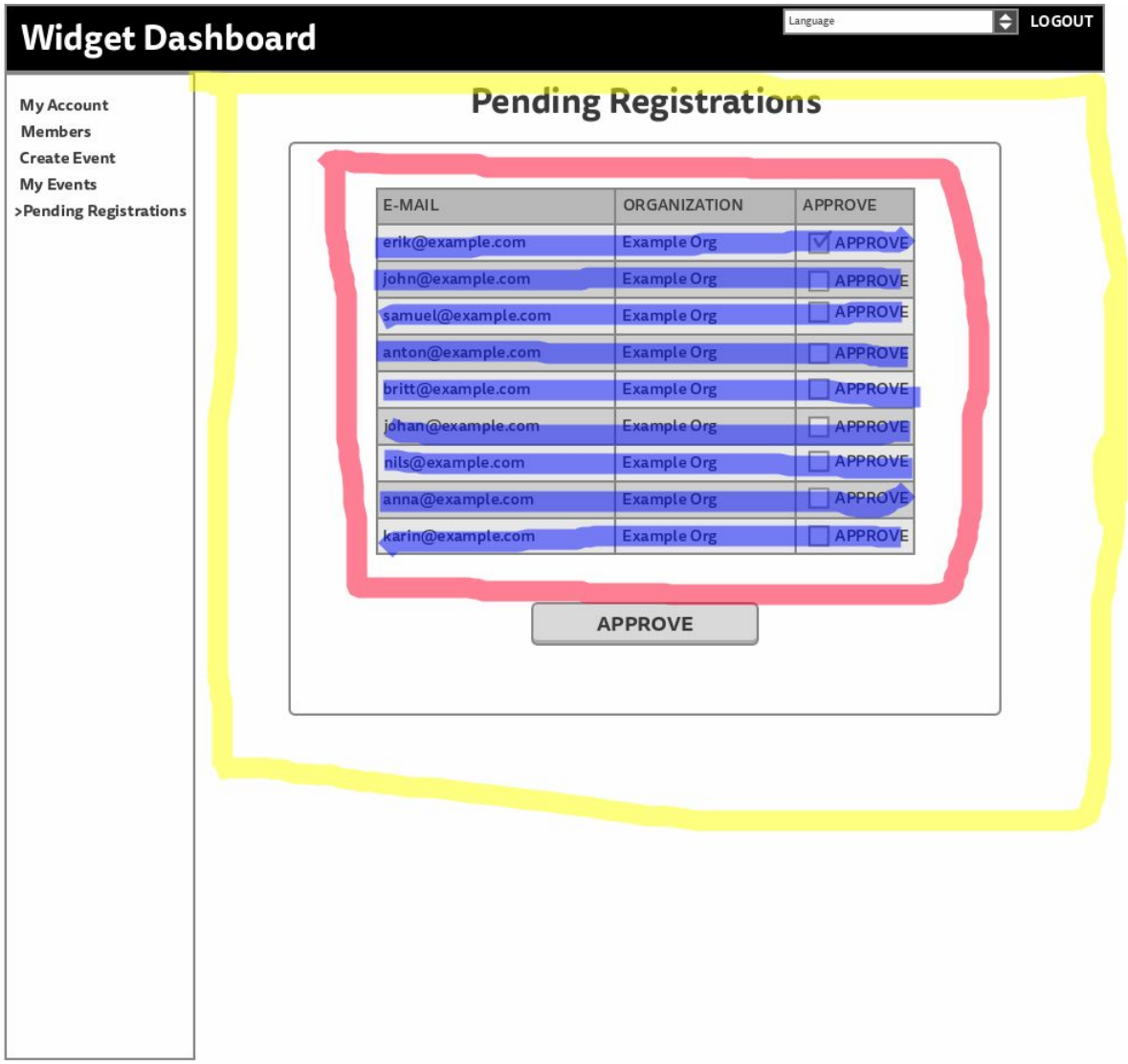
Green: Event Component. Responsible for rendering the event item in the list.

States:

events: Array - Store all the events requested from the server
toDelete: String - Stores the ID of an event that should be deleted.

popup: Object - Store the configuration for the popup which will be sent to ConfirmMessage Component.

PendingRegistrationsView



Yellow: PendingRegistrationsView Component. Responsible for rendering the Pending Registrations Page.

Red: RegistrationsList Component. Responsible for rendering the list header and loop out the right amount of Registration component.

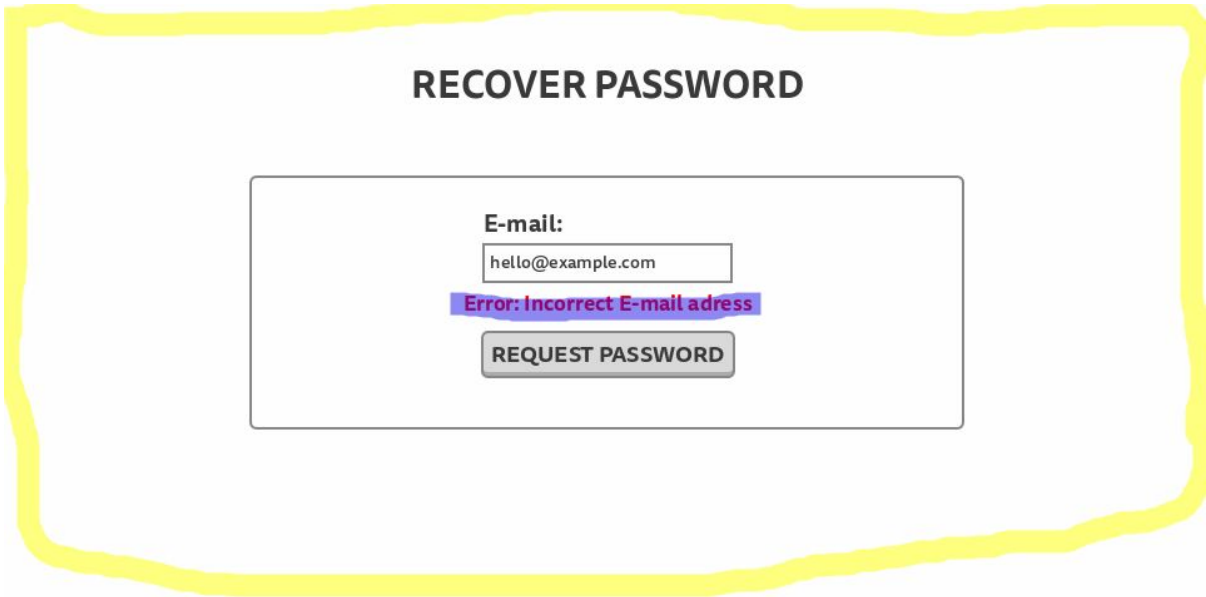
Blue: Registration Component. Responsible for rendering the registration item in the list.

States:

registrations: Array - Store all the pending registrations requested from the server
approve: Array - Keeps track of all the approve actions taken by the user.

deny: Array - Keeps track of all the deny actions taken by the user

RecoverView



Yellow: RecoverView Component. Responsible for rendering the Recover Password Page.

Blue: ErrorList Component. Responsible for rendering error messages if there are any.

State:

fields: Object - Stores all the input data from the user.

RegisterView

Yellow: RegisterView Component. Responsible for rendering the Register Page

Blue: Captcha Component. Responsible for rendering Google reCaptcha.

Green: ErrorList Component. Responsible for rendering error messages if there are any.

States:

fields: Object - Stores all the input data from the user
organizations: Array - Stores all the organizations requested from the server
newOrg: String - Keeps track on if the user wants to input a new organization or not
fieldErrors: Array - Stores all the error messages to be rendered.
_loading: Boolean - Keeps track on if a loading symbol should be shown while waiting for a response from the server.
_redirect: Keeps track on if the user has registered and should be redirected to the Login Page.

UpdatePasswordView

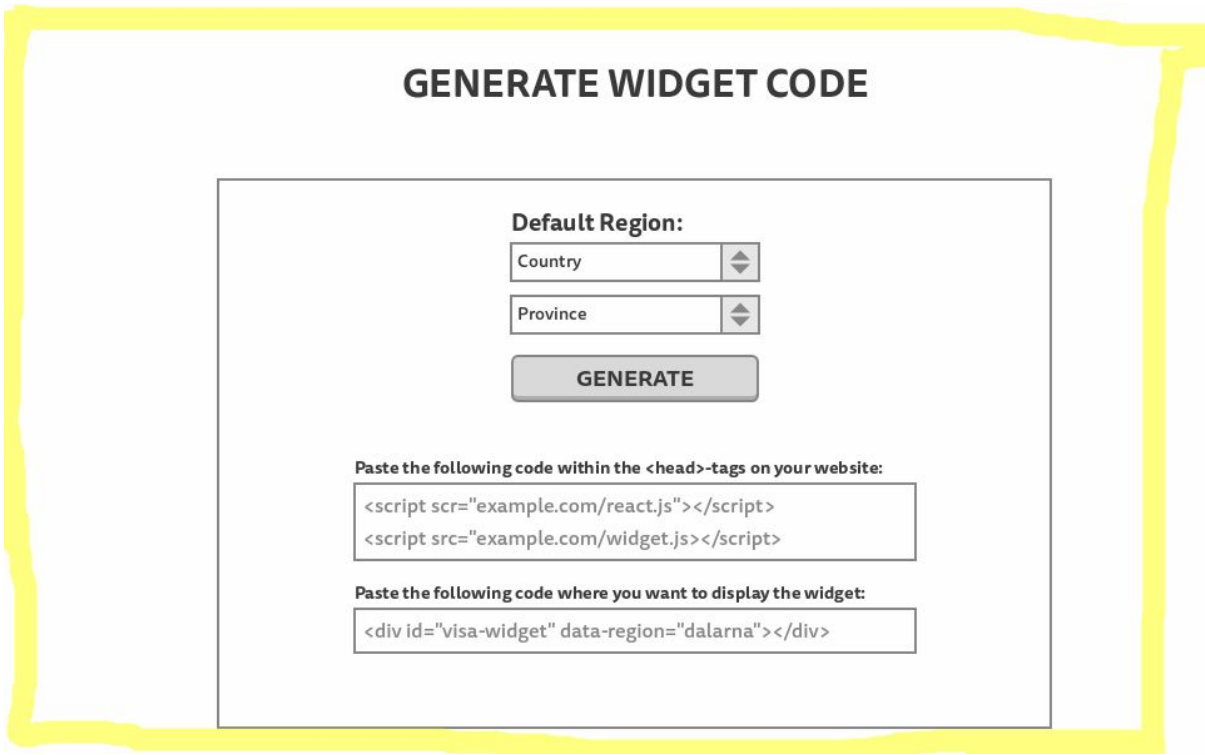
Yellow: UpdatePasswordView Component. Responsible for rendering the Update Password Page.

Blue: ErrorList Component. Responsible for rendering error messages if there are any.

States:

fields: Object - Stores all the input data from the user.

WidgetView:



Yellow: WidgetView Component. Responsible for rendering the Generate Widget Code Page.

States:

fields: Object - Stores all the input data from the user.
fieldErrors: Array - Stores all the error messages to be rendered.
isGenerated: Boolean - Keeps track if the widget code is generated or not.
headCode: String - Stores the headCode when generated.
bodyCody: String - Stores the bodyCode when generated.
token: String - stores the API token requested from the server.

Server

Overall

Because we are using the Spring framework it will dictate the higher levels of the architecture. For example, the controllers are not linked to the application as a normal dependency or attribute, but instead uses annotations which Spring uses to identify components. Restrictions like this will constrain our choice of architectural decisions.

User Package

The user package is there to handle all request which has to do with accessing or modifying anything to do with the users. The entry points for the package is the four controllers

VisitorController, UserController, AdminController, and SuperAdminController. These are linked to the main Spring application though the @RestController annotation.

The VisitorController handles any requests which can be performed by a non authenticated user, while the UserController handles any requests which can be performed by a logged in user, the AdminController takes care of any requests which should only be carried out by an administrator, and same for the SuperAdminController which handles request only allowed by a super administrator.

The reason for this separation is because Spring Security sets restrictions based on the URLs. So what we have done is map a URL to each controller and then restrict access to them based on which authorities the current user has. This grouping of functionality based on authority is a good way of minimizing mistakes in mapping the security.

Class	Request Mapping	Required Authorities
VisitorController	/api/visitor	-
UserController	/api/user	USER
AdminController	/api/admin	USER, ADMIN
SuperAdminController	/api/super_admin	SUPER_ADMIN

The UserDAO uses the strategy pattern to make it easy to change the DAO class to one that uses another type of database without having to change too much code. For our implementation we will only have one implementation of this which is the UserDAOMongo which has an implementation to fit MongoDB. The UserDAO interface is used by all controllers within the package, and while you could make a singleton solution here there is no need since the DAO object is stateless.

Next we can look at the User class which is there to represent the persistent User in memory. It contains two other packages AuthenticationLink and Organization. The structure of these three is driven by the design of the database. They are made so we can easily map a stored user to a User object and then have that user in memory. The User package also have some methods for checking the user on login, but also two methods for comparing it to another user. One for checking if this user can manage parts of the other user, and the other one is if this user can change role of another user, for example which him/her to administrator.

One problem when working with a static typed language and APIs is that the User object does not always fit what we want to send or receive from the front end. Therefore we need “Data Transfer Objects” or DTOs. They are kept in a sub package of User, but is treated as a separate package, because of how Java works. Because of this they need to be public, but it would be to messy to have them in the user package. But these are used to map the information being sent and received in the controllers through http calls.

Other small packages are UserInformationValidator which takes different DTOs and validates the information in them. Often used with registration or changes of password. Then we have Email package which takes an email address and an id and sends a special link to the email address. It's used for validating email and recovery of password. CurrentUser takes a static method in spring and just makes it easier to mock in testing. And AuthorizationChecker is a facade towards the other packages where they can send in a user id to check if the currently logged in used can manage that other user with the matching id.

Event Package

The Event package is responsible for handling all requests to which deals with events.

Class	Request Mapping	Required Authorities
EventController	/api/event	USER (with exceptions)

The EventController handles all requests dealing with events. However one could argue that it should be split in two since parts like getting events is a publicly available service, while the rest of the controller requires the USER authority. But because it is only two exceptions to the authority I saw no need to split then at this point.

The DAO works in a similar way to how it's used in the User package where the strategy pattern can be used to easily change the DAO to one that works with a different DBMS.

The complicated part about the events in our system is that they are all location based. So the client needs to be able to fetch events in a certain area. For the we are using coordinates which are saved in the EventLocationCoordinates. But since coordinates are not what we humans use we are taking advantage of the Google Maps API to take addresses, area, and location and convert to coordinates. All other data we can find on the places are also stored in the EventLocation class. The EventLocation object is then contained in the Event object which represents the persistent object in memory.

Token Package

The Token package is a small package which only does one thing, saves a token in the database which is sent back to the created api token. The TokenController is a RestController with one method which is called with a GET request. To have as good interchangeability as the other packages we are using the strategy pattern for the DAO. And the only method this has is to add the API token to the database.

There is also a class called TokenValidator which is used by other packages like the event package to check if incoming tokens are valid or not.

Class	Request Mapping	Required Authorities

TokenController	/api/token	-

DatabaseConnections Package

The purpose of the server is to more or less shuffle data between the client and the database, which makes the database connection a large dependency in the server. It needs to be accessible from everywhere in a good way. The problem with database drivers is the they all look very different which makes it hard to create good abstractions. Instead we created this package to house classes with static methods for accessing the database connection or client.

So when someone wants to use a different database with our system they would have to create new DAOs which is easily done thanks to the strategy pattern. But they would also have to create a new class in this package which has the driver for the new database.

There is no interface or rules on how these classes should look since they are only implemented in the DAOs anyways, which are themself easily interchangeable. Another factor is the vast differences between database drivers and how they handle connections. Some drivers like the MongoDB driver handles connection pools for you, while others don't which would mean in a decently large system you would have to implement that yourself.

The big downside for this is that it makes mocking and therefor testing quite a bit harder since dealing with static dependencies in testing can be tricky. But the ease of use and interchangeability makes that a well deserved sacrifice.

DatabaseServices Package

Since these were small services which ran independently from the rest of the application I wanted to make it as easy as possible to add or remove any service from the application.

I created the Service abstract class which has two abstract methods getname() when implemented will get the name of the service, and the stop() method which prepares the service for stopping so the thread can be terminated. Service also implements the Runnable interface which has the run() method which is where the functionality of the service will run. All services will extend this class so they will all be identified as a Service.

To remove most complexity from running several services in different threads I used to Facade Pattern and created the ServiceManager class. On this you can perform three basic actions, add a service, start all services, and stop all services. This of course uses the Service class i talk about in the previous paragraph.

Image Package

Spring already handles most of the complexities of static content handling, so the most important part of designing static content handling was image handling. Thankfully, the hardest parts are already included in Spring. Automatic image compression with gzip and filesize limits are available by changing the application settings. We decided not to reinvent

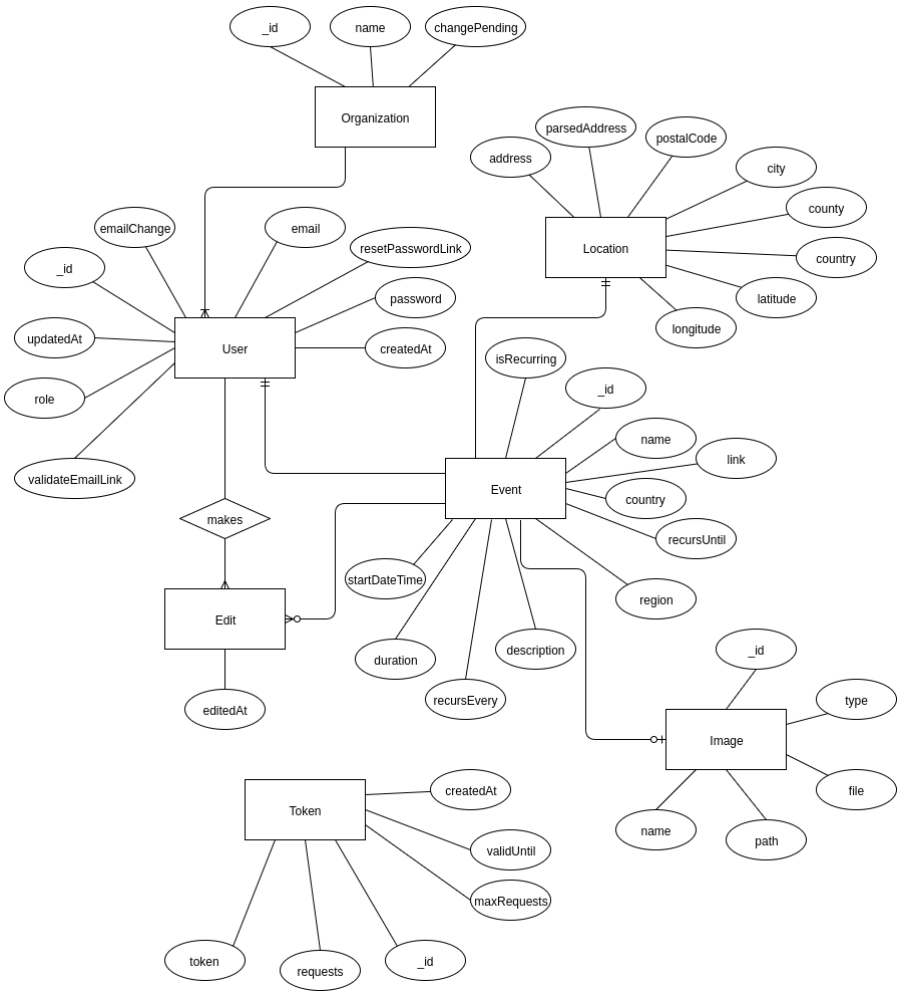
the wheel, especially considering the tight deadlines we are under, and use these in-built tools.

That leaves image uploading, storing images in the database, and retrieving them for display. Since we do not need a separate class for handling static files such as HTML, CSS and JavaScript, we can create a single class that handles images that we called ImageHandler. ImageHandler is responsible for uploading images to a GridFS MongoDB storage, saving a reference to the stored image in the relevant Event document, and later retrieving the image when necessary. We chose to use GridFS because this lets us store documents larger than 16MB in case this is necessary. We will make sure to rename the images in order to avoid conflict between images with the same name, then store the image reference on the Event document as a URL pointing to the new image name.

The ImageHandler class contains public methods for storing, retrieving, and deleting images from the database, as well as a DAO interface and its MongoDB implementation. This DAO is responsible for updating events when necessary.

Database

In designing the database, since we decided to use MongoDB for reasons outlined in the design document, we had to decide whether to normalize the data or not. That is to say, we had to decide whether to embed documents in other documents or whether to store these relationships as references.



We decided to store objects as references rather than embed them in their “parent” documents. First, we stored User documents as references in the Organization objects. This is because an organization might

have hundreds of users and this makes the organization documents smaller and more easy to manage. We will also frequently access the User objects on their own, which would make storing them as objects on the Organization document impractical.

As for the event object, we first decided to store these as documents embedded in the User objects. This was because each user only has a few events at most and that would mean it is not impractical to store these on the User document as they will not take up too much space. We eventually changed this to storing the events as references in the User documents as the event documents will frequently be accessed on their own and it would be impractical to loop through every User document to find every Event document for viewing in the widget.

However, Events are associated with users and it would be practical to have them stored as embedded documents for when a user is accessing his or her dashboard. Another solution would be to store the events as embedded documents and on their own, but this would entail duplication and might complicate keeping the data consistent. If the cost of searching through the events documents to find a user's documents is too high compared to the cost of updating both a user's embedded Event document and the separate Event object it might be worth considering this alternative.

Schema

Iterations

Iteration 1

Iteration 1 consisted of the basic schema of the database, including which entities had to be modelled and what attributes and types each entity had to contain. In this iteration, we created the three basic entities User, Event, and Organization and gave them attributes. In iteration 1, design decisions had to be made regarding if documents should be embedded into other documents or a reference stored in that document. We decided to embed events in the User entity and store a reference to users in the Organization entity. This is because each Organization might have hundreds of users, but each User will only have a few or dozens at most events.

Iteration 2

In this iteration, we added the Token entity. This entity is responsible for storing the request tokens for the calendar API. These may or may not be attached to a specific User account, since even unregistered users can generate widget code that includes an API token. On the event entity, we changed locationCoords and locationDesc to a single object. We added the createdBy attribute on the Event object and changed User to store EventID rather than embed the events on the user object.

Iteration 3

In order to align the database more with how Spring works, we changed isAdmin and isSuperAdmin from a boolean to a role attribute which is a String. We also changed the way images are stored in the Event object from a binary object to a URL String. This keeps the size of the event documents down and lets us handle the implementation of image handling separately, while also making it easy to retrieve the images when an event is viewed. We also changed the embedded User documents in the Event document to UserID, so that we don't have to store too much unneeded information on the Event document.

Iteration 4

In this iteration, in order to solve the problem of needing to check whether a user was approved by their organization or not, we decided to remove the Organization collection entirely and instead make it an attribute on the User document. Since we do not need to access information about organizations when it's not in relation to a user, we consider this change to make it faster and cheaper to access relevant data about organizations. The organization object on the User contains a changePending boolean that determines whether a user is currently waiting to change organizations.

Iteration 5

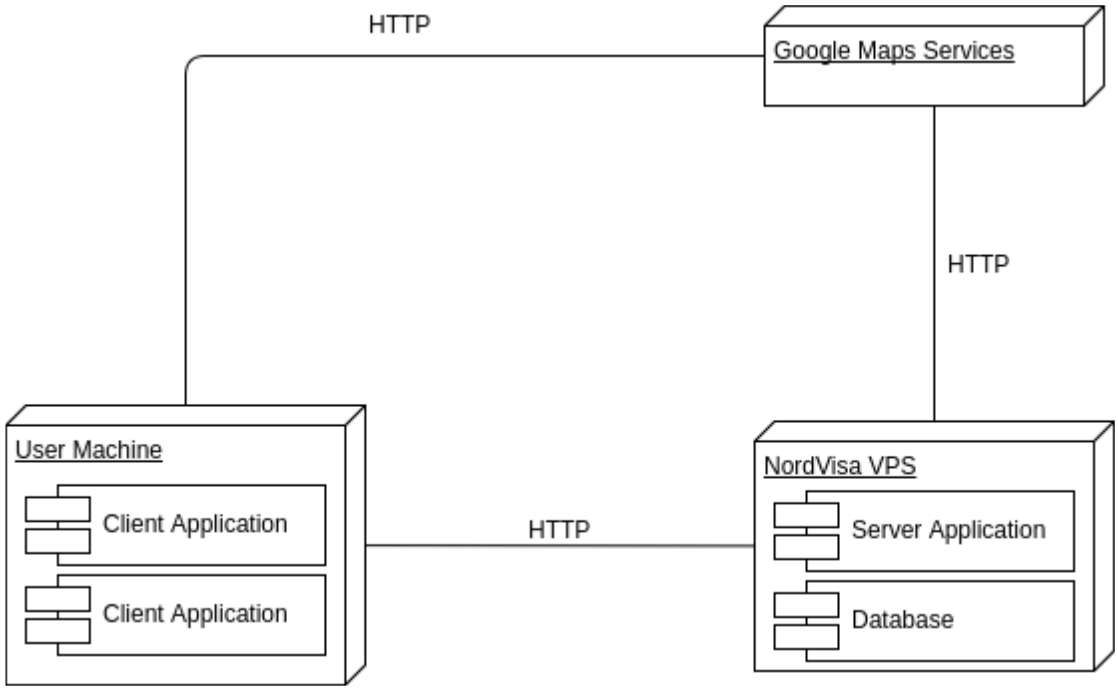
This iteration adds the Image entity to the database and removes the relationship between User and Token.

Schema Table

User	_id: ObjectID email: String password: String role: String resetPasswordLink: { url: String, timestamp: Datetime } validateEmailLink: { url: String, timestamp: Datetime } createdAt: long updatedAt: long emailChange: String organization: { name: String, approved: Boolean, changePending: String }
Event	_id: ObjectID name: String location: { coords: [Float, Float], address: "", parsedAddress: "",

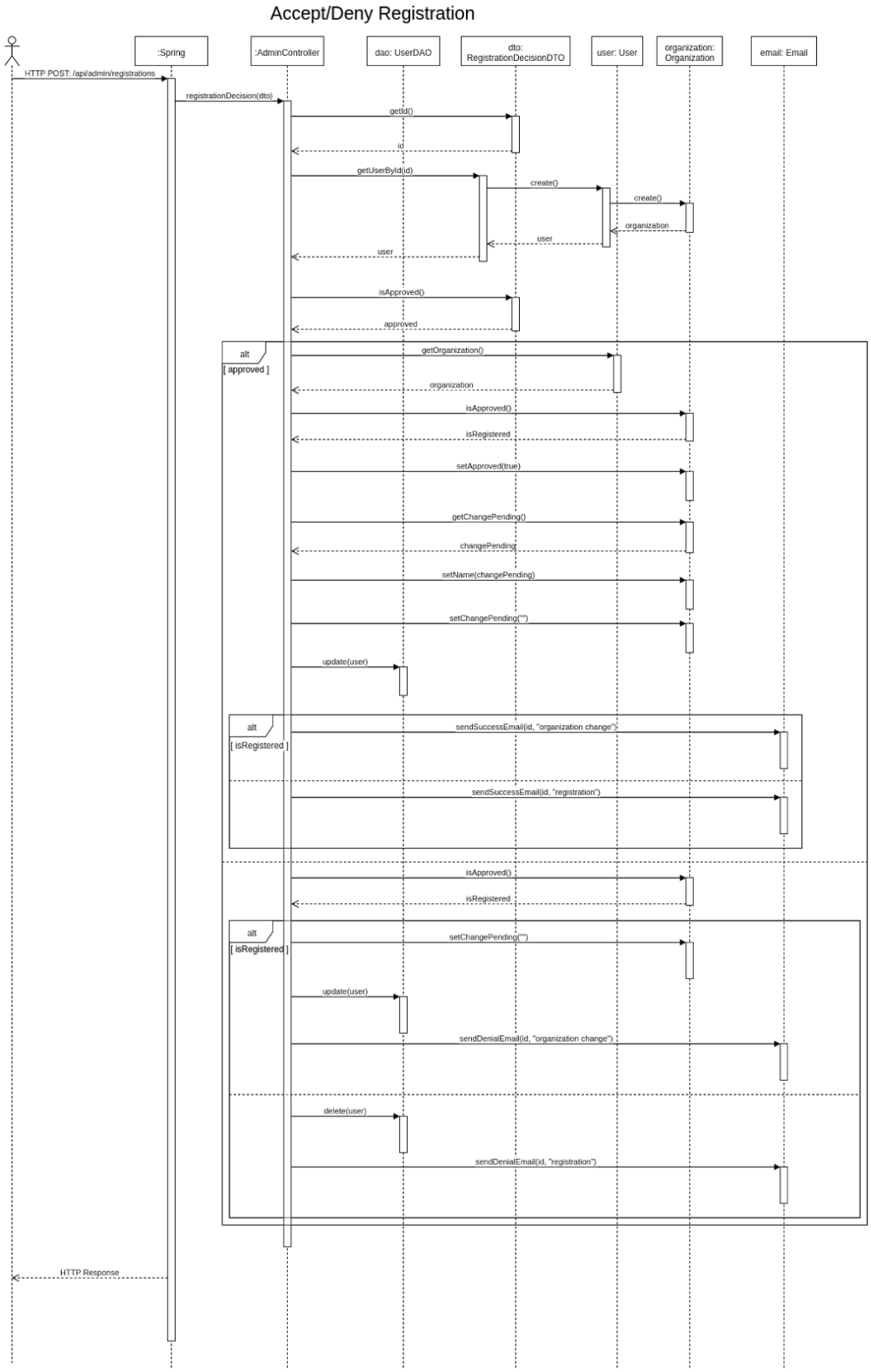
	postalCode: “”, city: “”, county: “”, country: “” } description: String startDateTime: long duration: long recurring: Boolean recursEvery: String recursUntil: Date images: [String, String, String] createdAt: long updatedAt: long createdBy: String editedBy: String path: String
Image	_id: ObjectID name: String path: String file: Byte[] type: String
Token	_id: ObjectID token: String requests: Integer maxRequests: Integer createdAt: long validUntil: long

Appendix I: Deployment Diagram

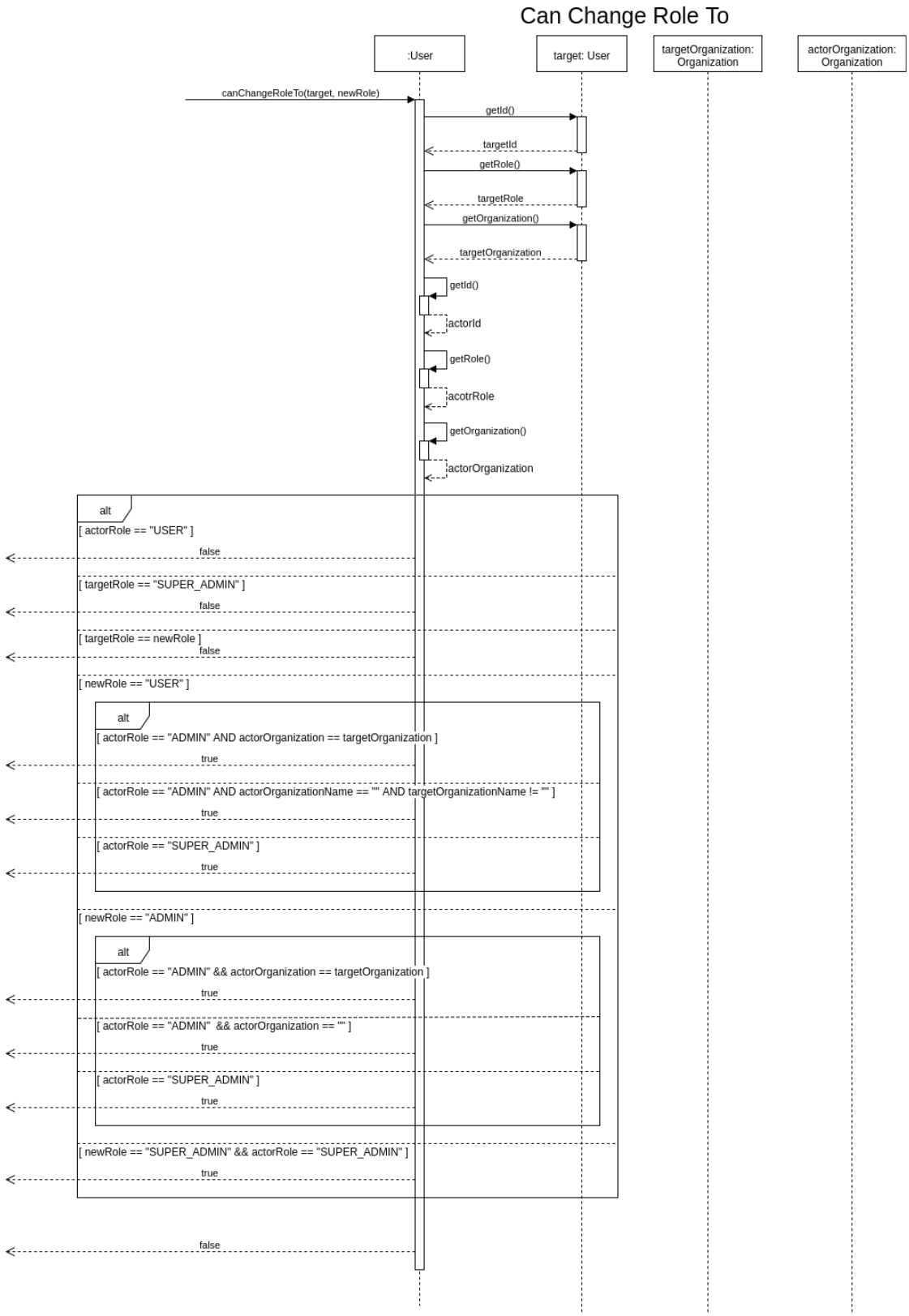


Appendix II: Sequence Diagrams

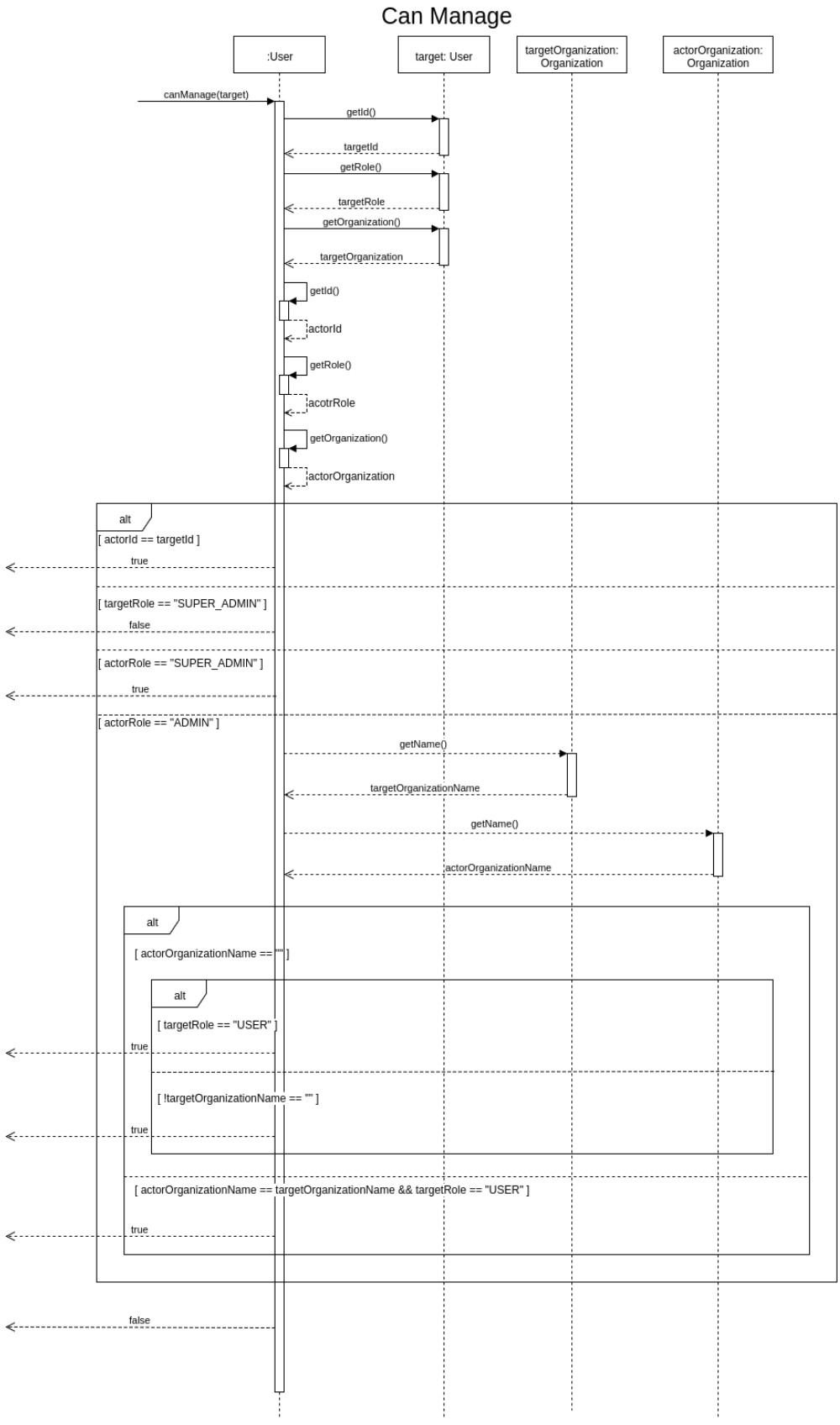
Accept/Deny registration



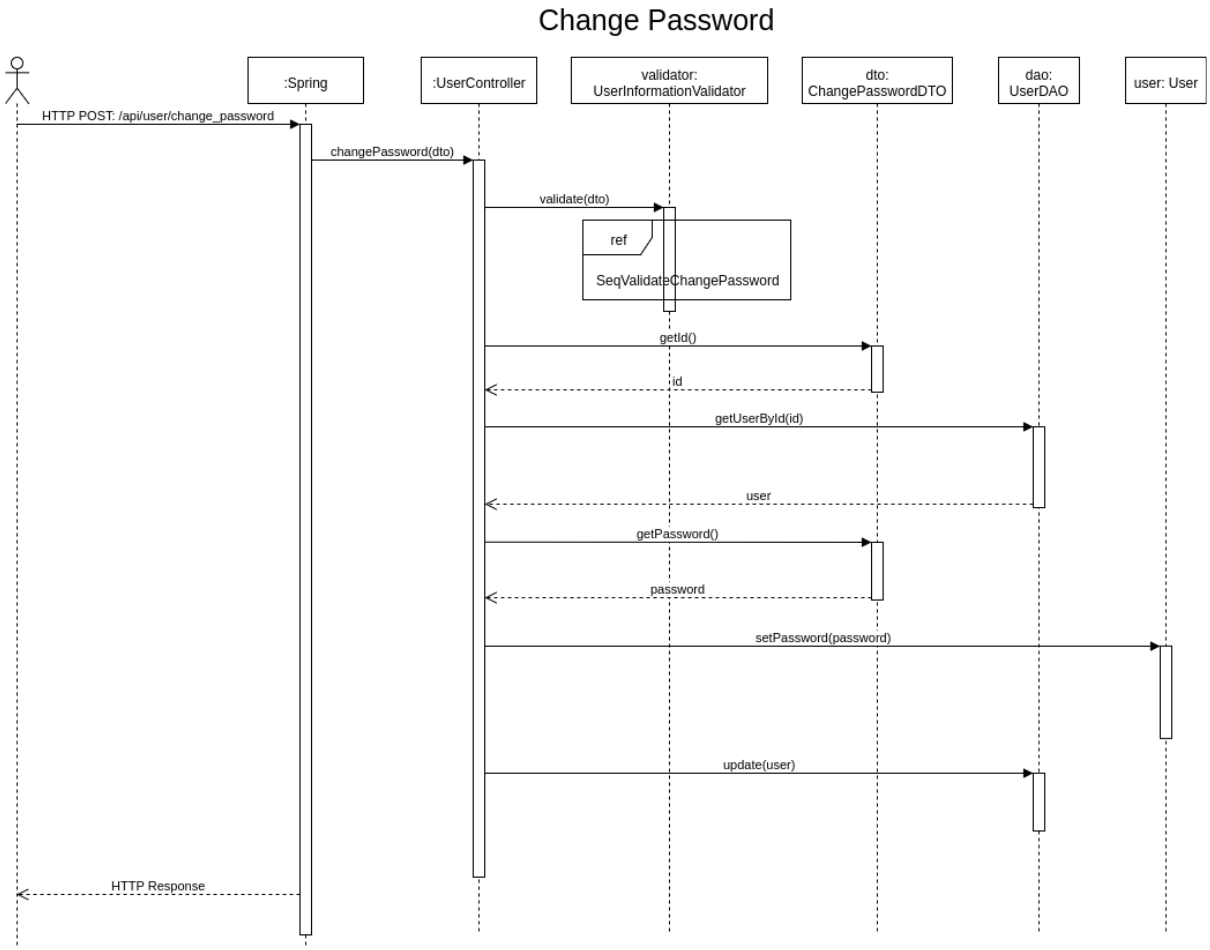
Can Change Role of Other User



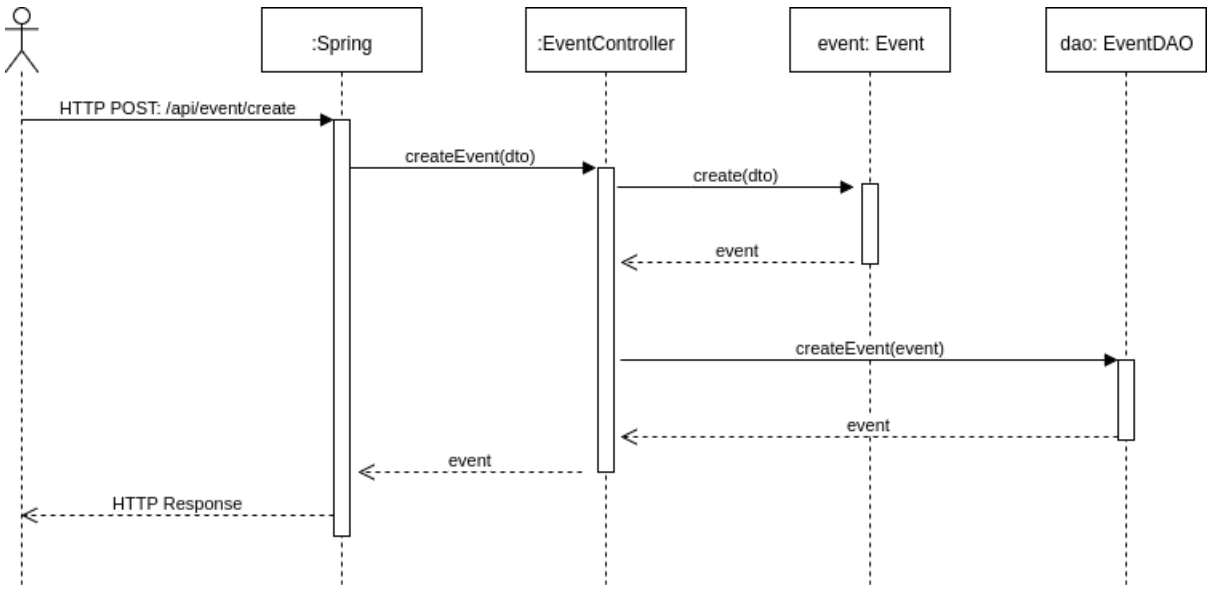
Can Manage Other User



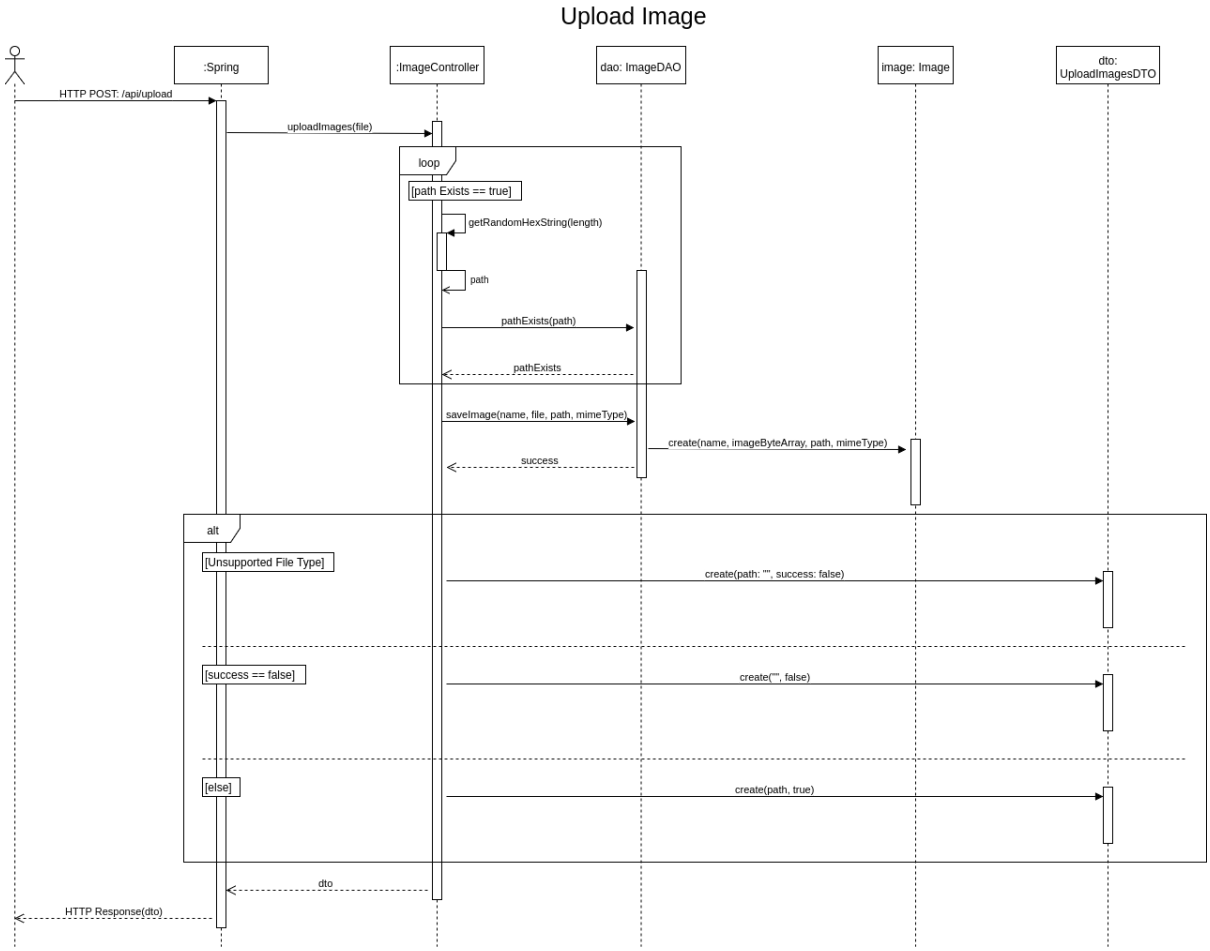
Change Password



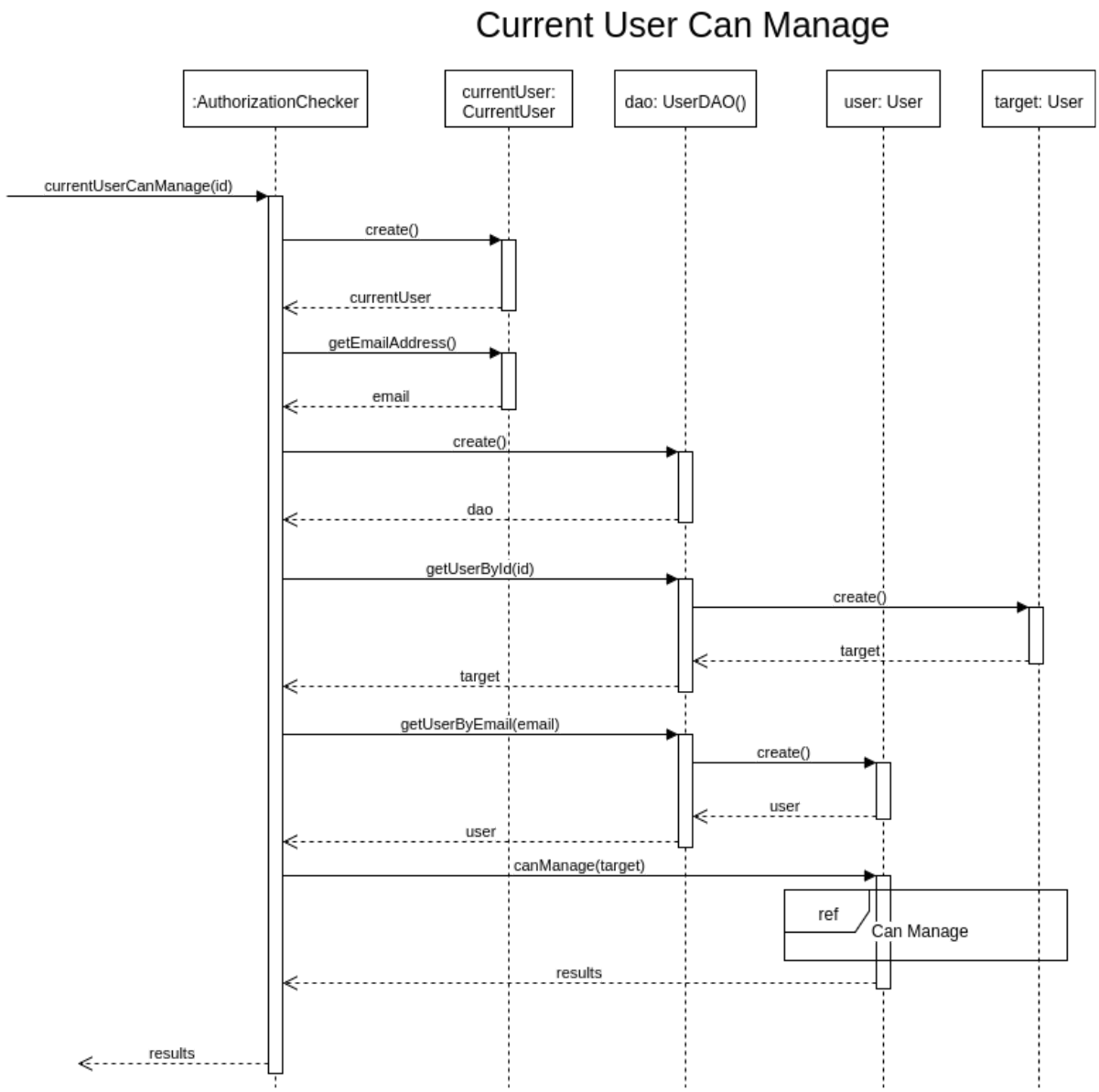
Create Event



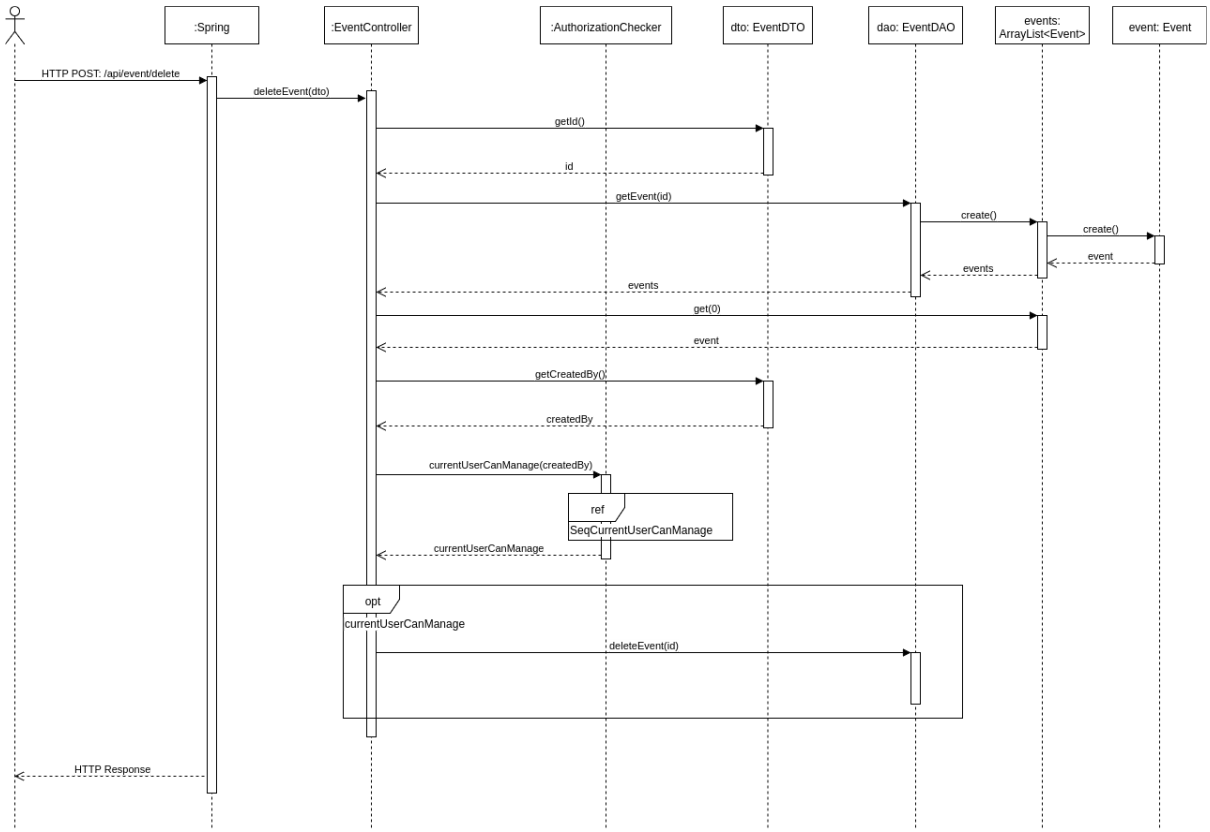
Create Image



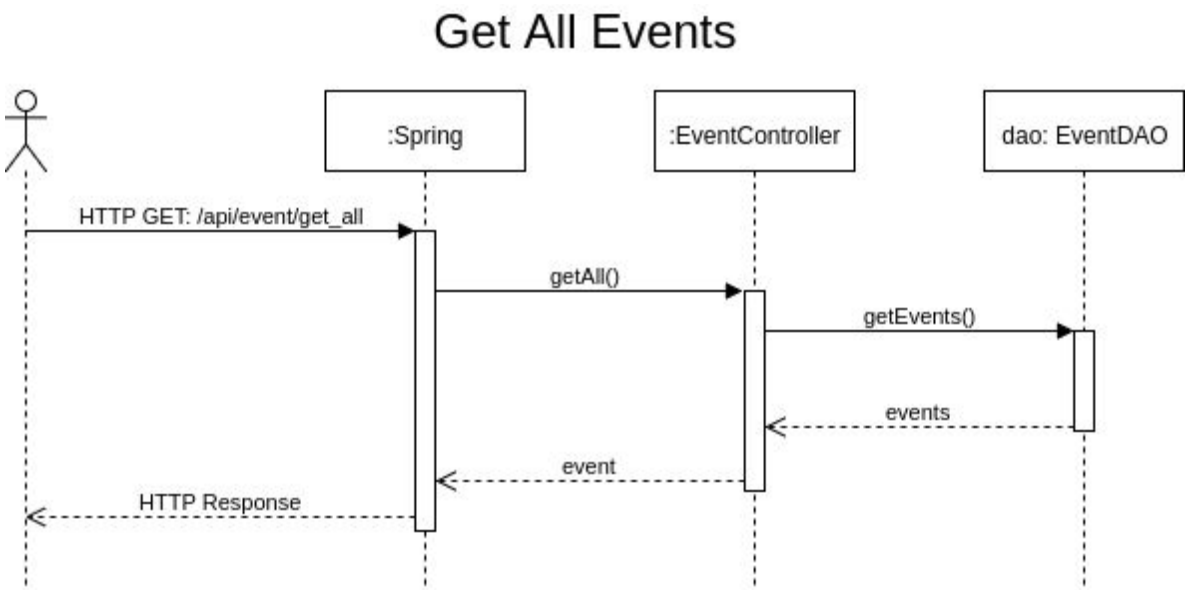
Current User Can Manage



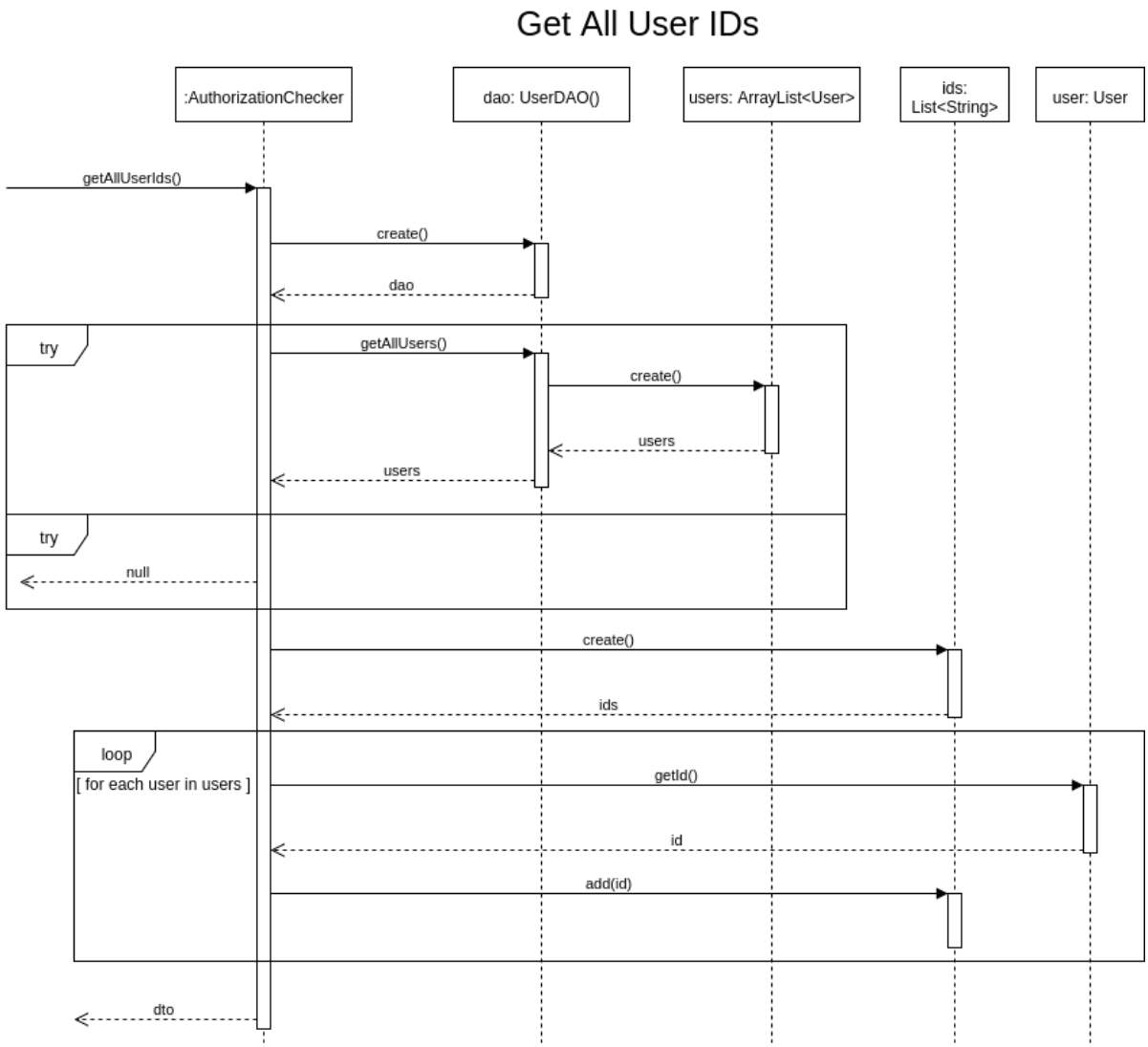
Delete Event



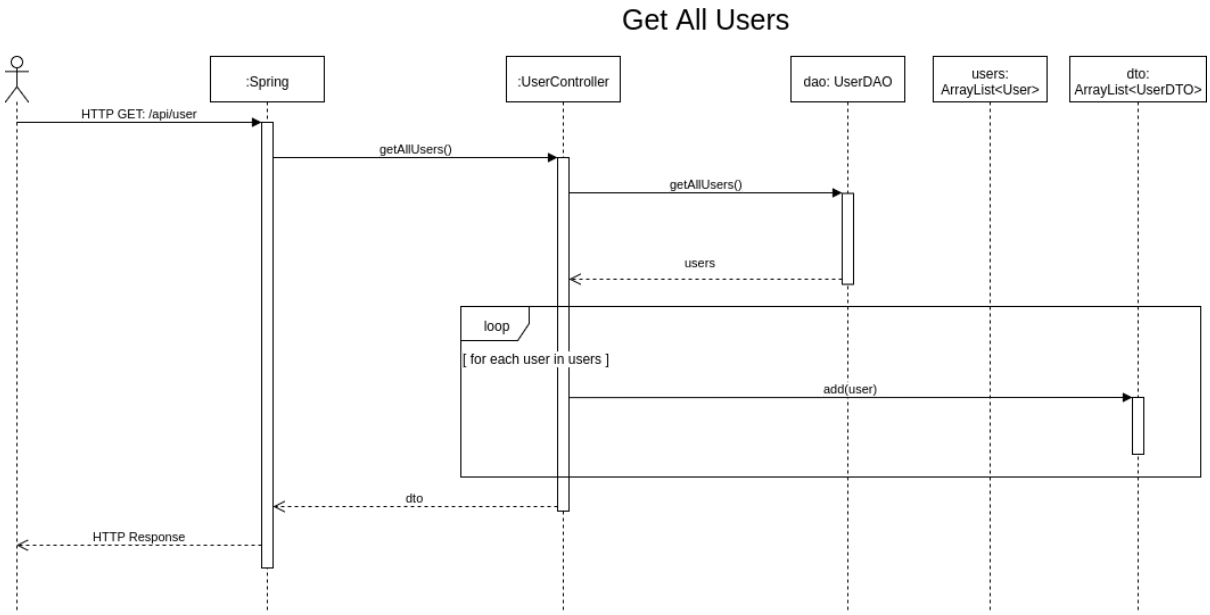
Get All Events



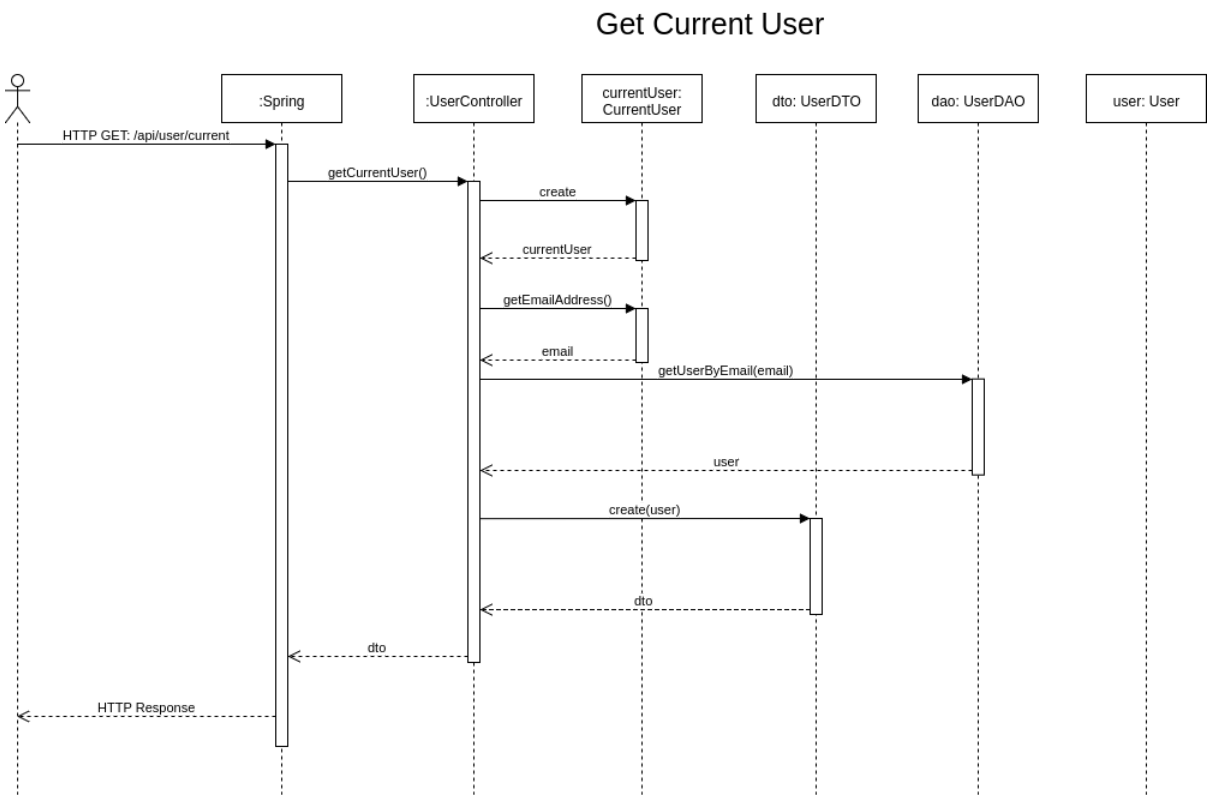
Get All User IDs



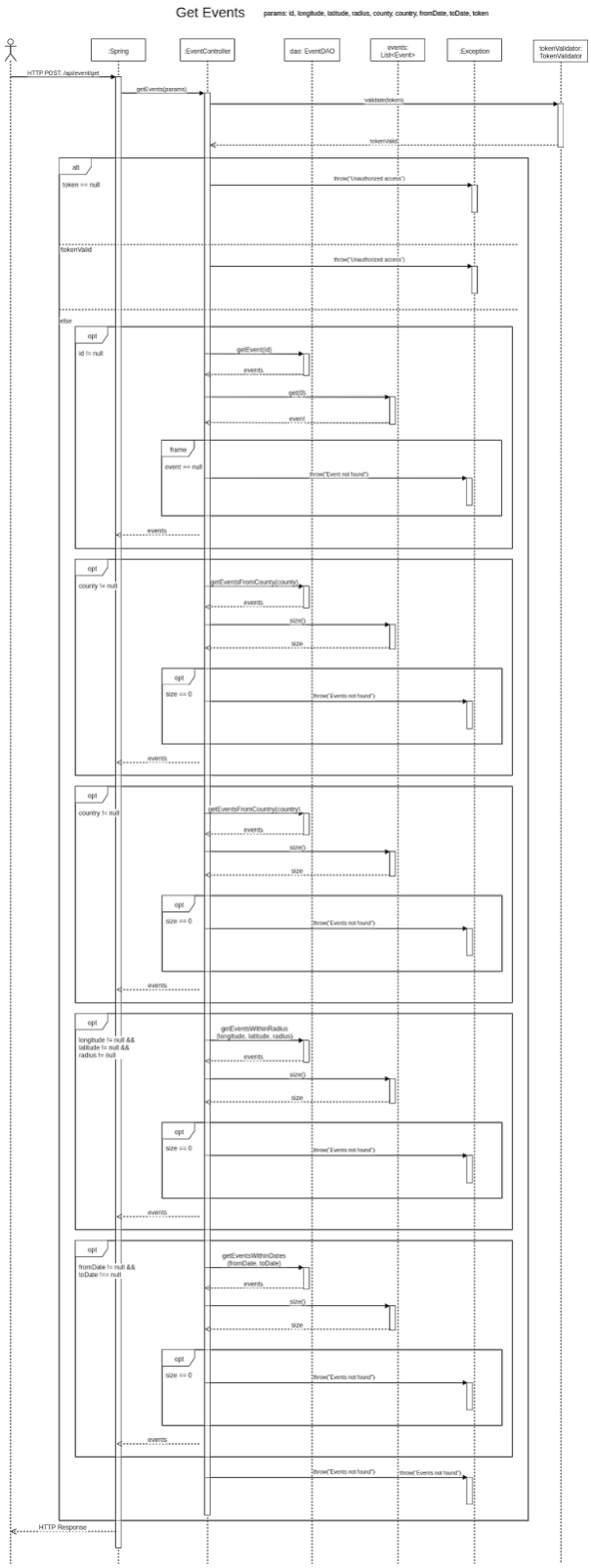
Get All Users



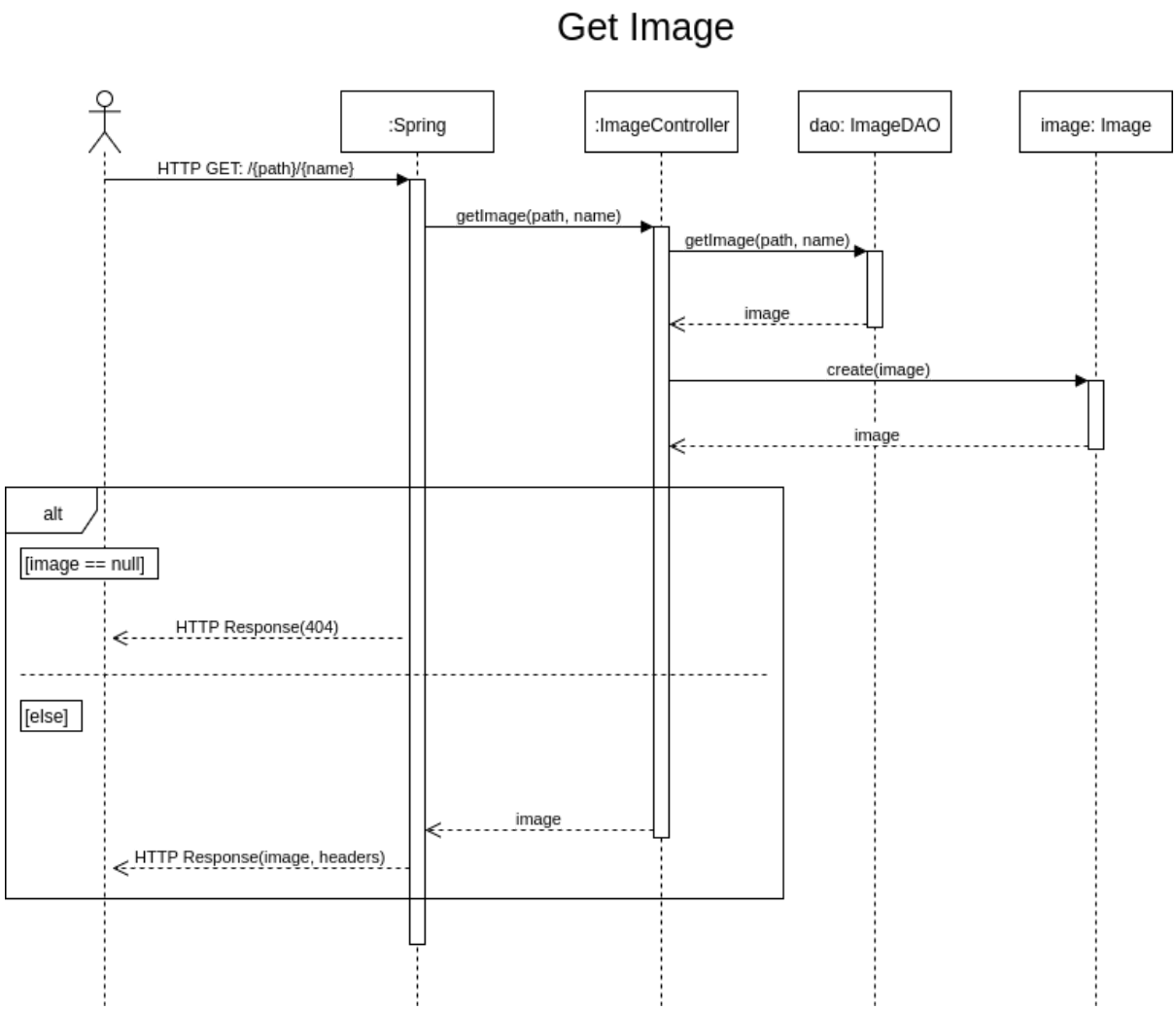
Get Current User



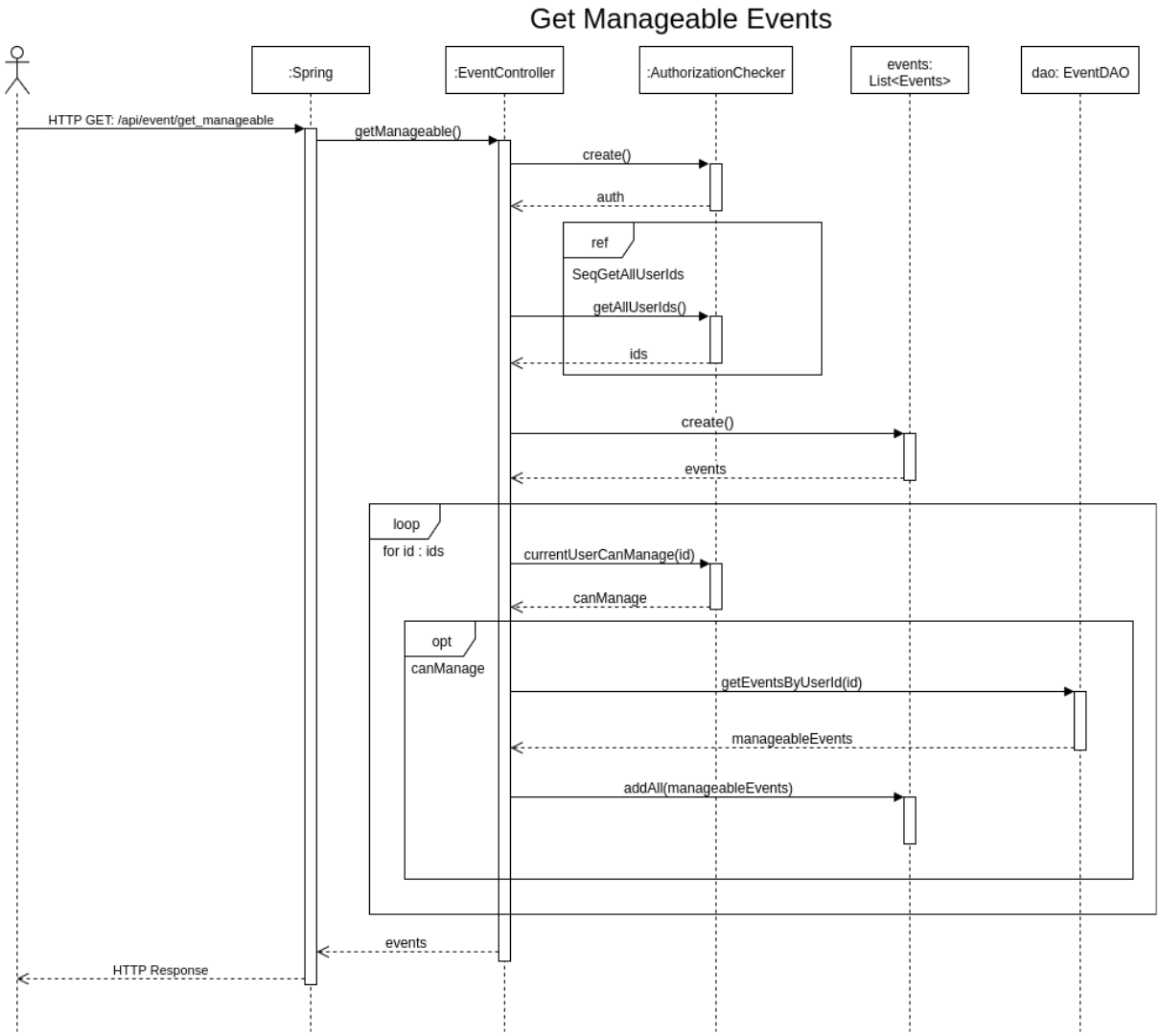
Get Events


Get Events
params: id, longitude, latitude, radius, county, country, fromDate, toDate, token
:Spring
:EventController
dao: EventDAO
events: List<Event>
:Exception
tokenValidator: TokenValidator
HTTP POST: /api/event/get
getEvents(params)
validate(token)
tokenValid
alt
token == null
throw("Unauthorized access")
tokenValid
throw("Unauthorized access")
else
opt
id != null
getEvent(id)
events
get(0)
event
frame
event == null
throw("Event not found")
events
opt
county != null
getEventsFromCounty(county)
events
size()
size
opt
size == 0
throw("Events not found")
events
opt
country != null
getEventsFromCountry(country)
events
size()
size
opt
size == 0
throw("Events not found")
events
opt
longitude != null &&
latitude != null &&
radius != null
getEventsWithinRadius
(longitude, latitude, radius)
events
size()
size
opt
size == 0
throw("Events not found")
events
opt
fromDate != null &&
toDate !== null
getEventsWithinDates
(fromDate, toDate)
events
size()
size
opt
size == 0
throw("Events not found")
events
throw("Events not found")
throw("Events not found")
HTTP Response

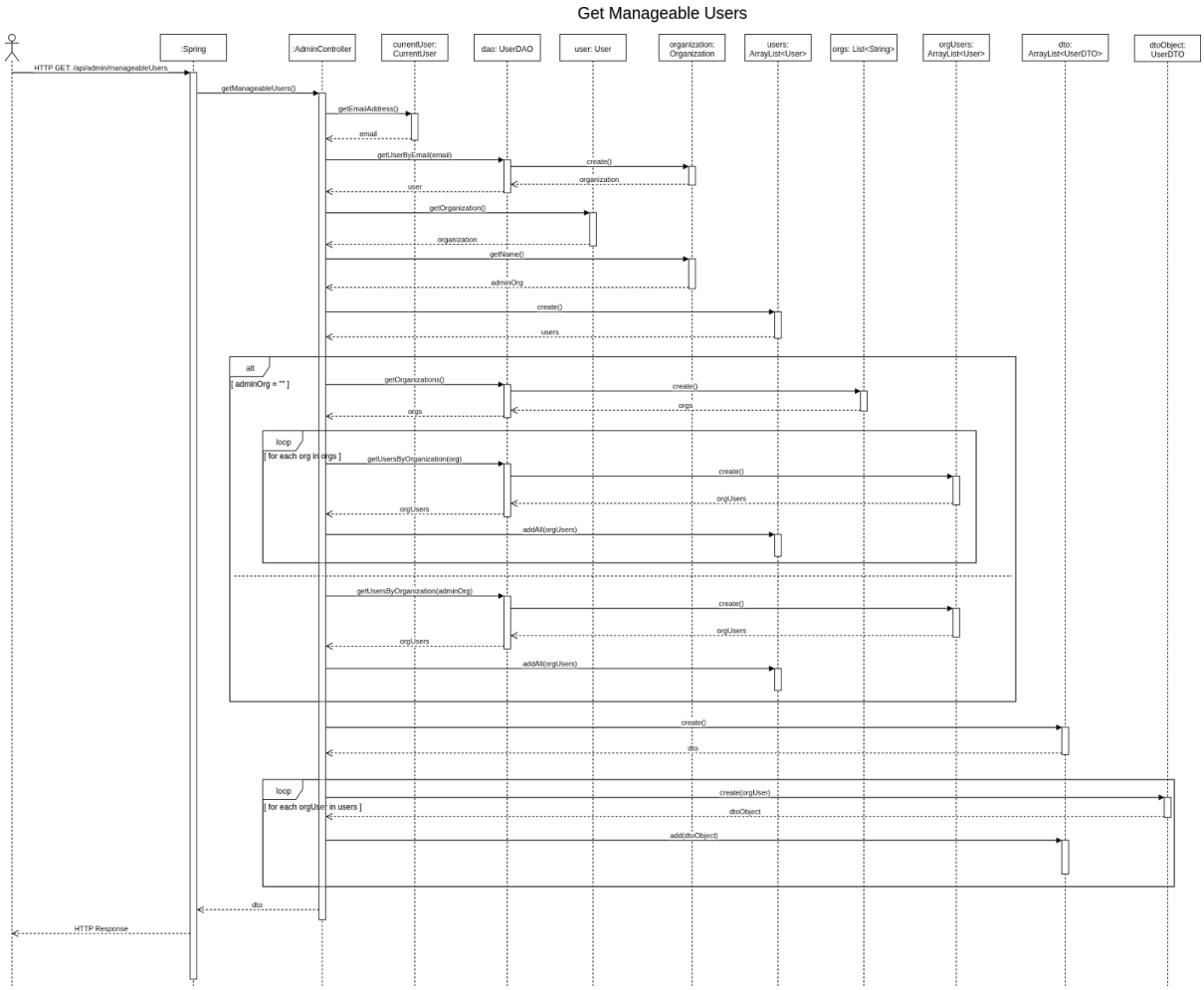
Get Image


Get Image
:Spring
:ImageController
dao: ImageDAO
image: Image
HTTP GET: /{path}/{name}
getImage(path, name)
getImage(path, name)
image
create(image)
image
alt
[image == null]
HTTP Response(404)
[else]
image
HTTP Response(image, headers)

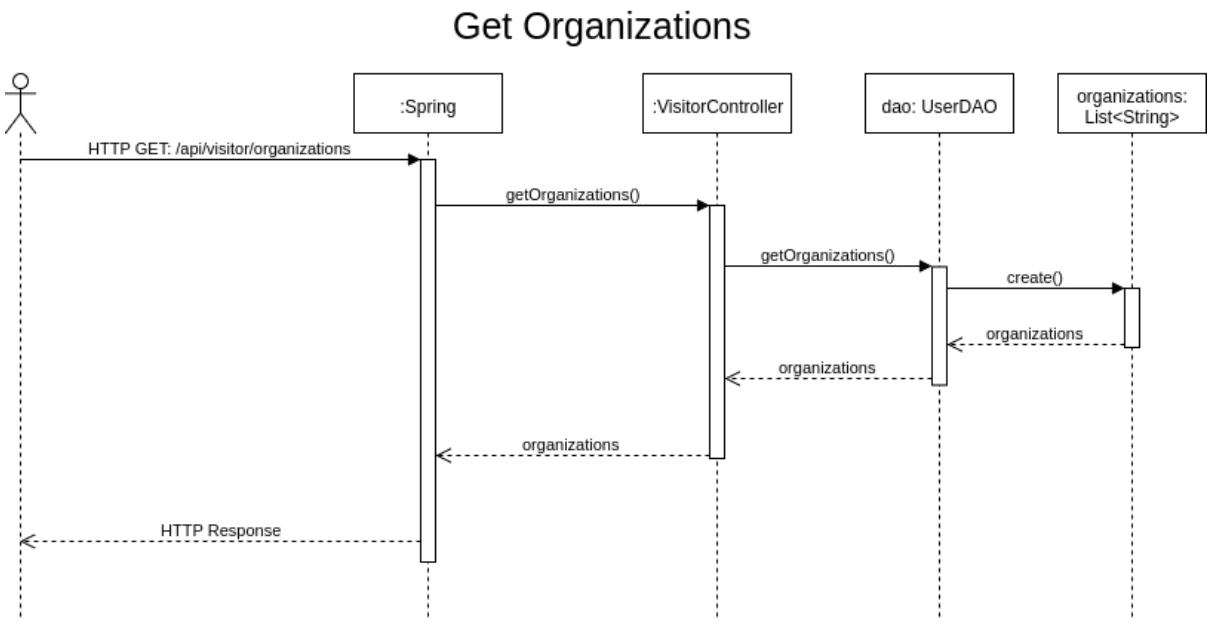
Get Manageable Events



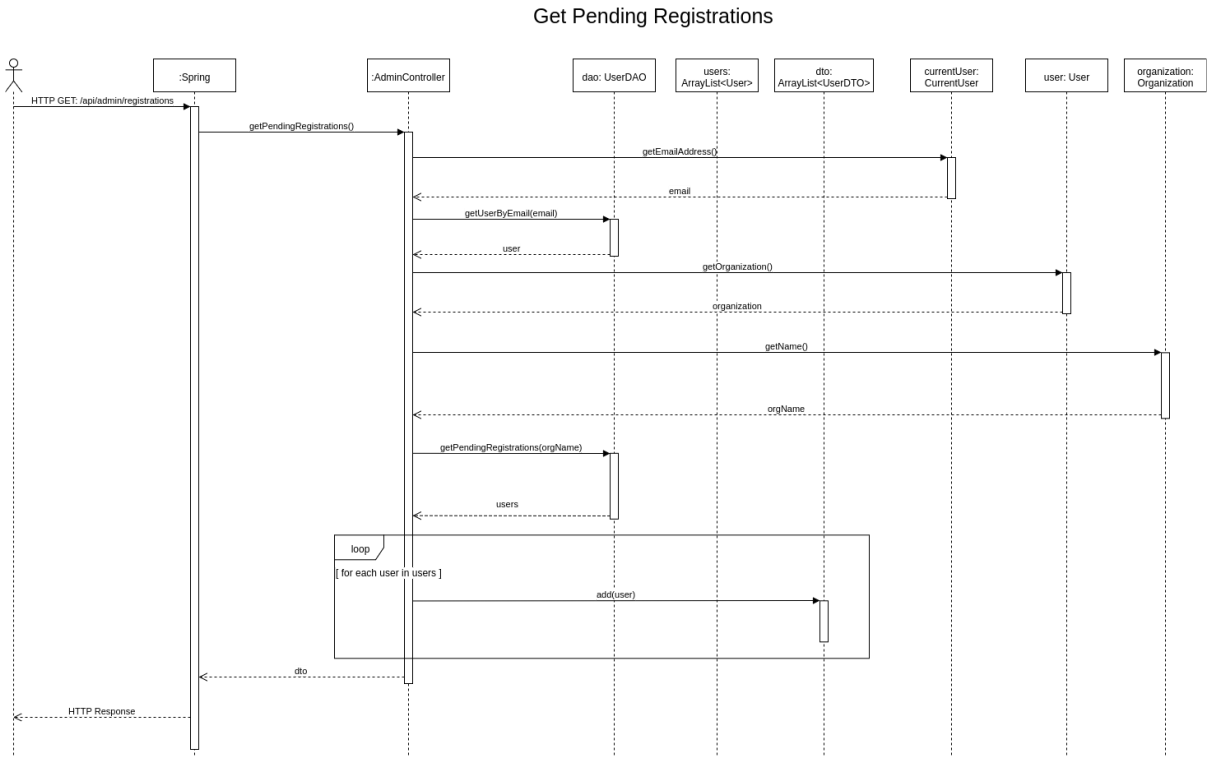
Get Manageable Users



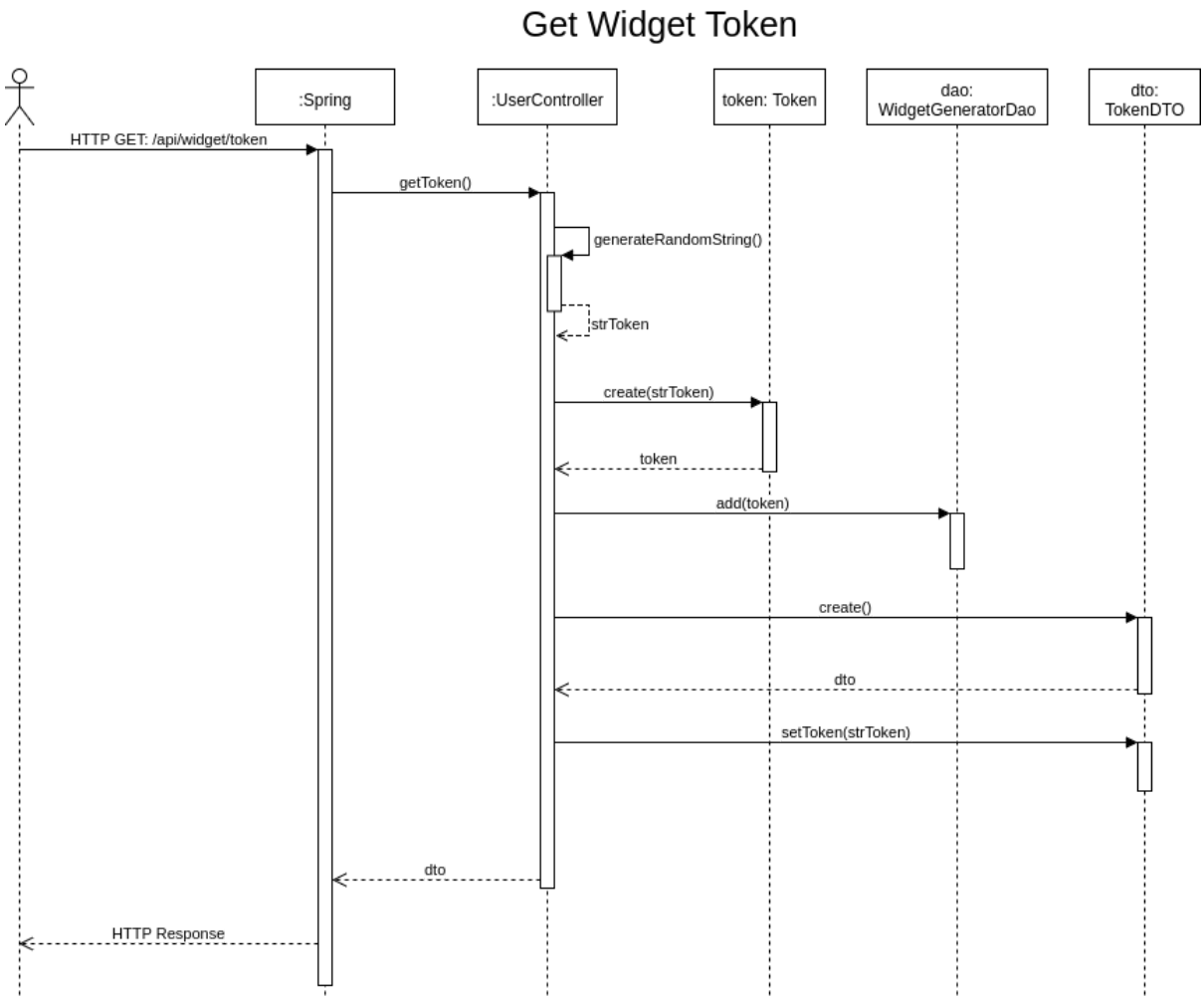
Get Organizations



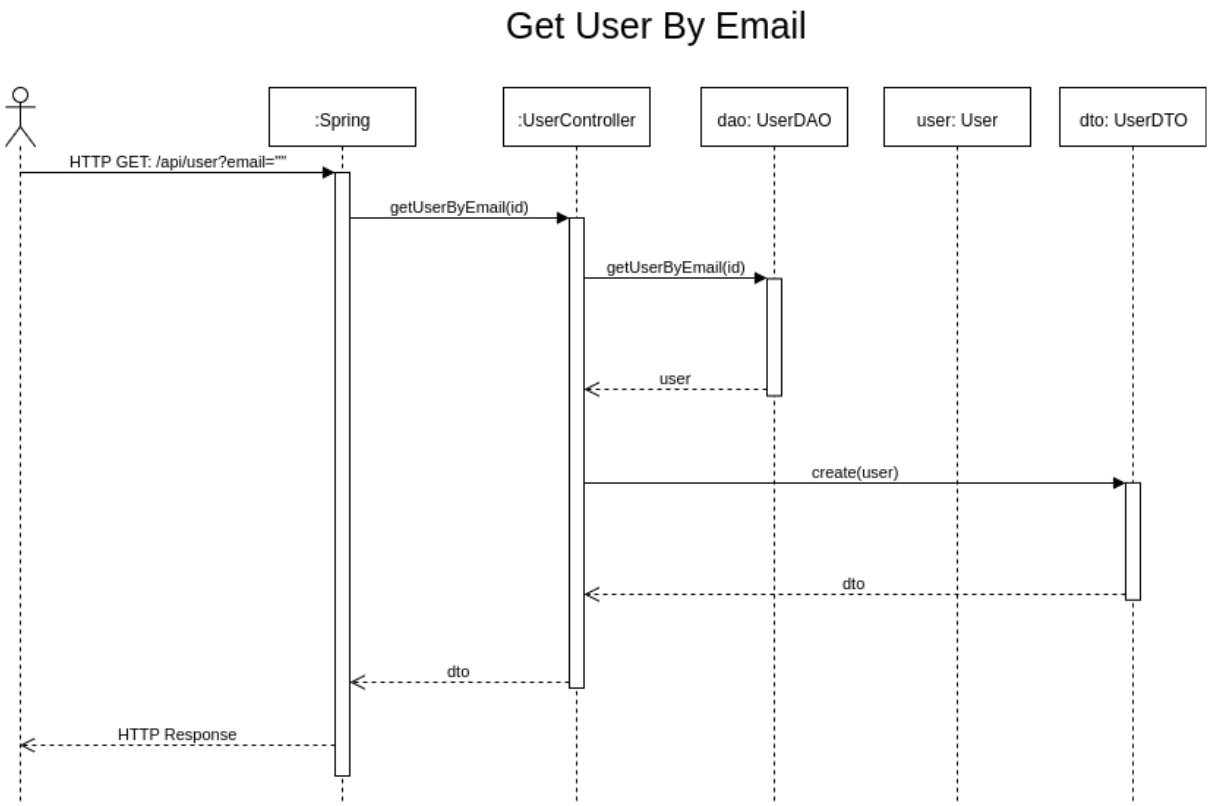
Get Pending Registrations



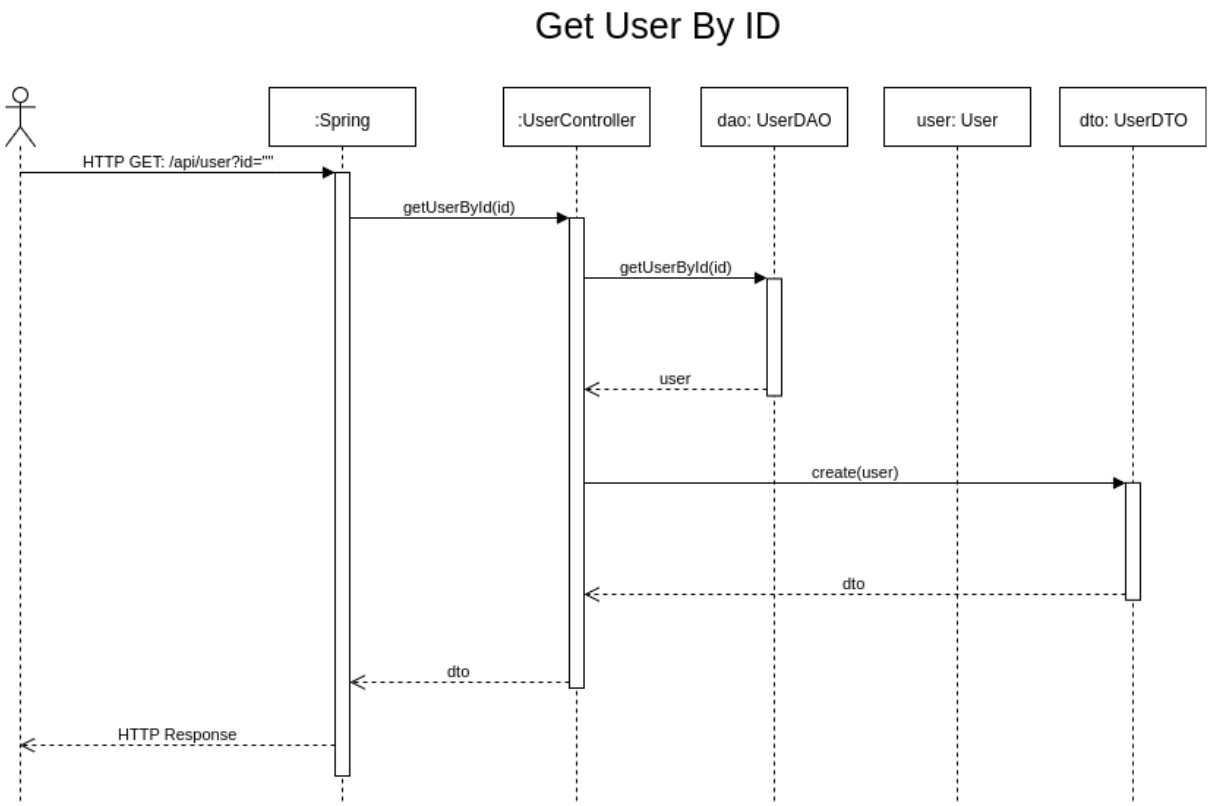
Get Token


Get Widget Token
:Spring
:UserController
token: Token
dao:
WidgetGeneratorDao
dto:
TokenDTO
HTTP GET: /api/widget/token
getToken()
generateRandomString()
strToken
create(strToken)
token
add(token)
create()
dto
setToken(strToken)
dto
HTTP Response

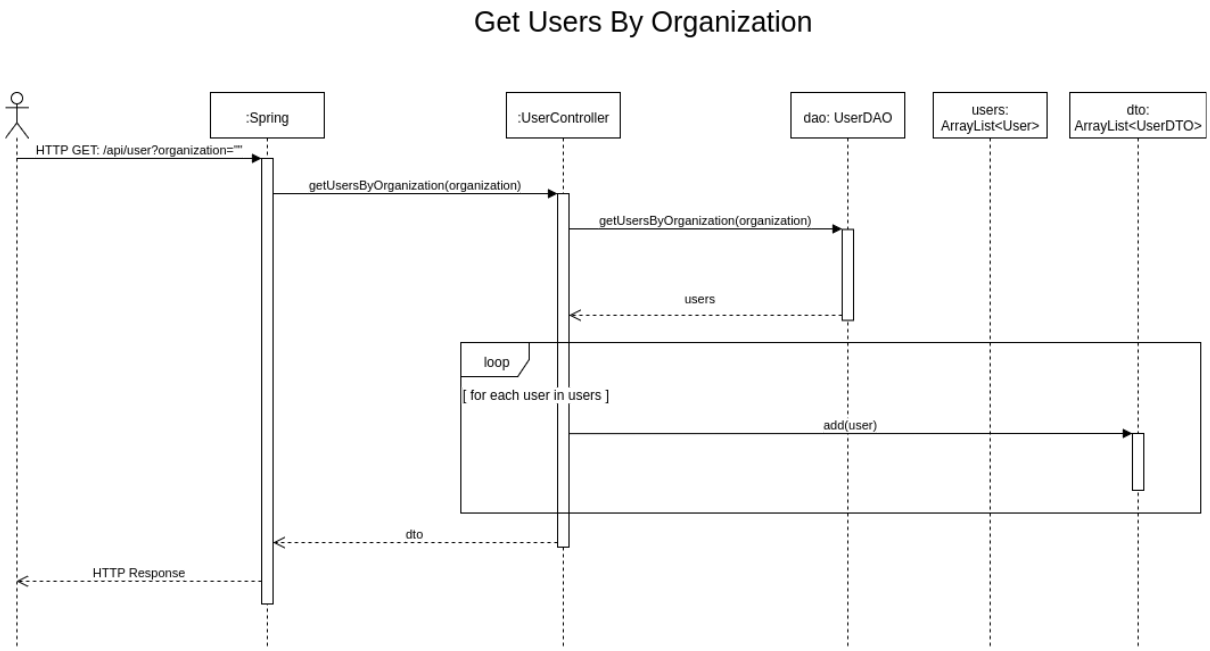
Get User By Email



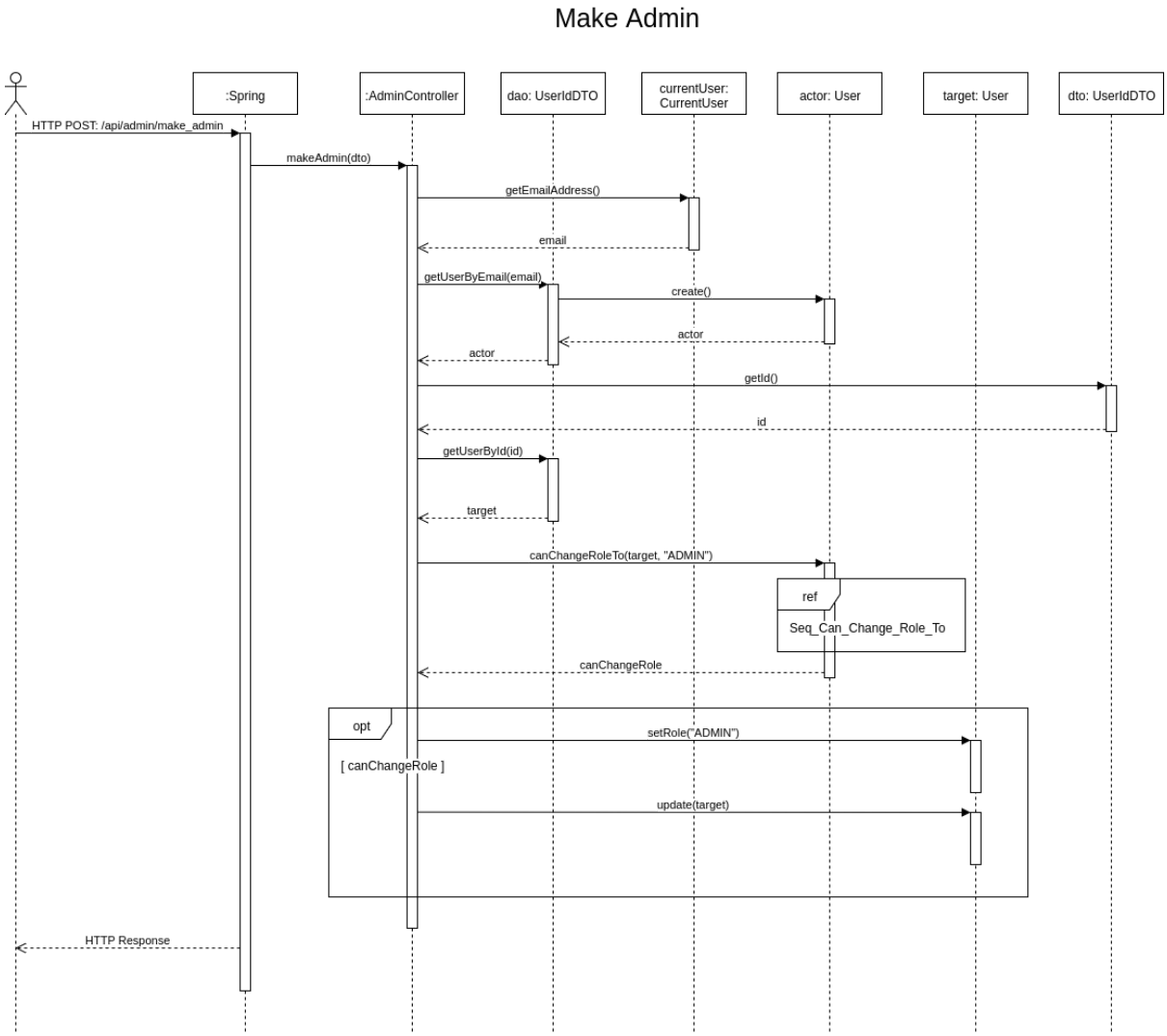
Get User By Id



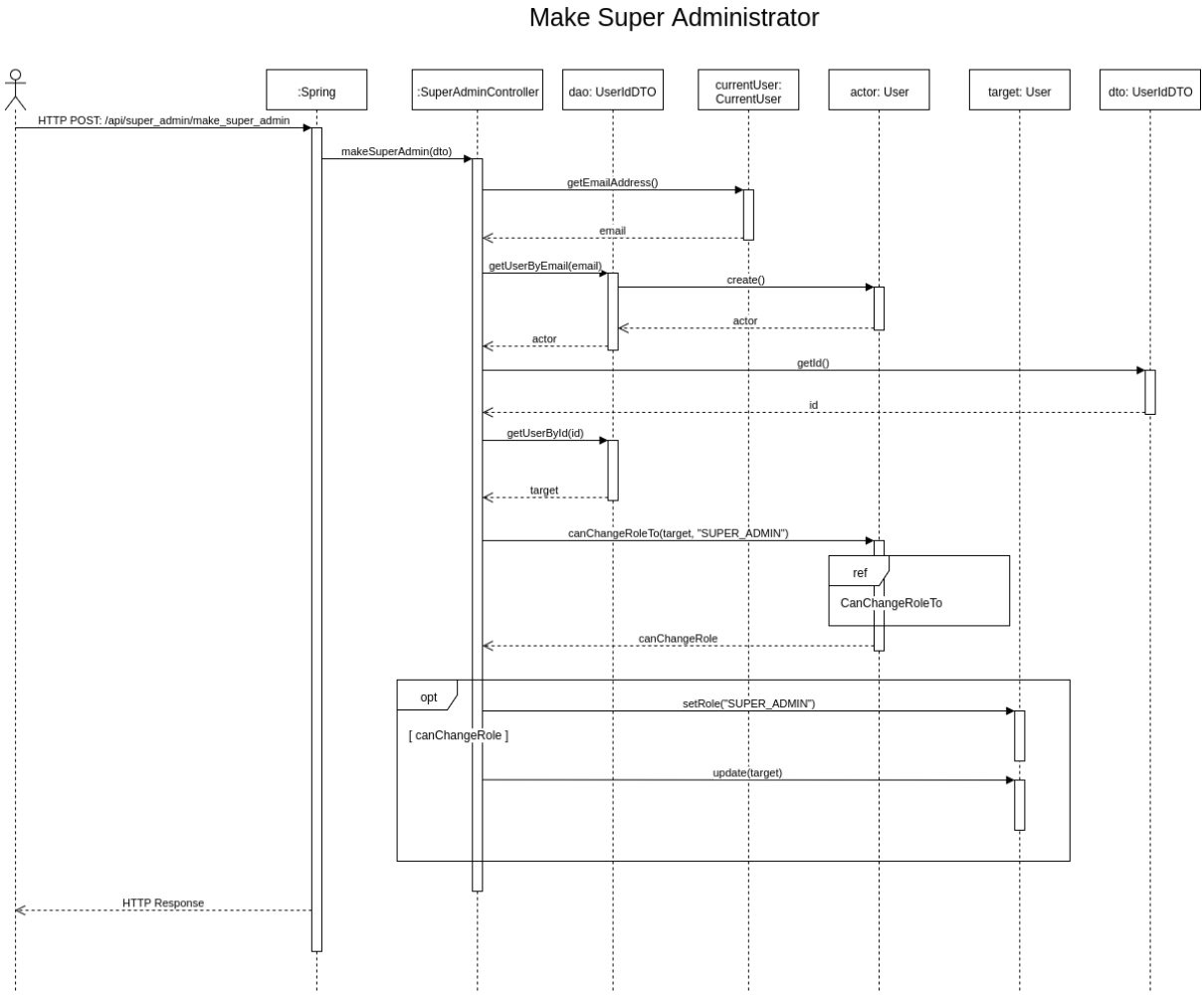
Get Users By Organization



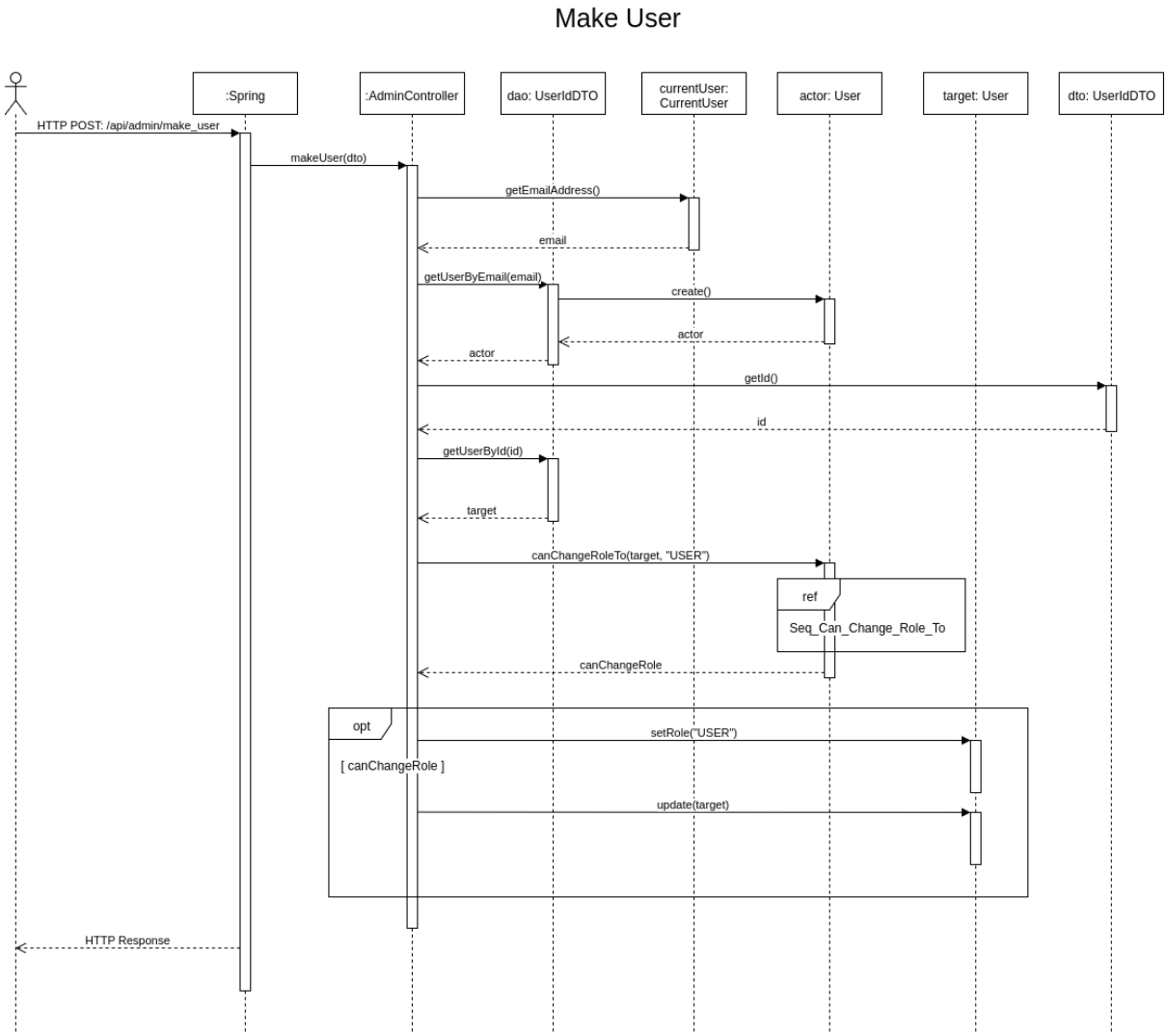
Make Admin


Make Admin
:Spring
:AdminController
dao: UserIdDTO
currentUser: CurrentUser
actor: User
target: User
dto: UserIdDTO
HTTP POST: /api/admin/make_admin
makeAdmin(dto)
getEmailAddress()
email
getUserByEmail(email)
create()
actor
actor
getId()
id
getUserById(id)
target
canChangeRoleTo(target, "ADMIN")
ref
Seq_Can_Change_Role_To
canChangeRole
opt
[canChangeRole]
setRole("ADMIN")
update(target)
HTTP Response

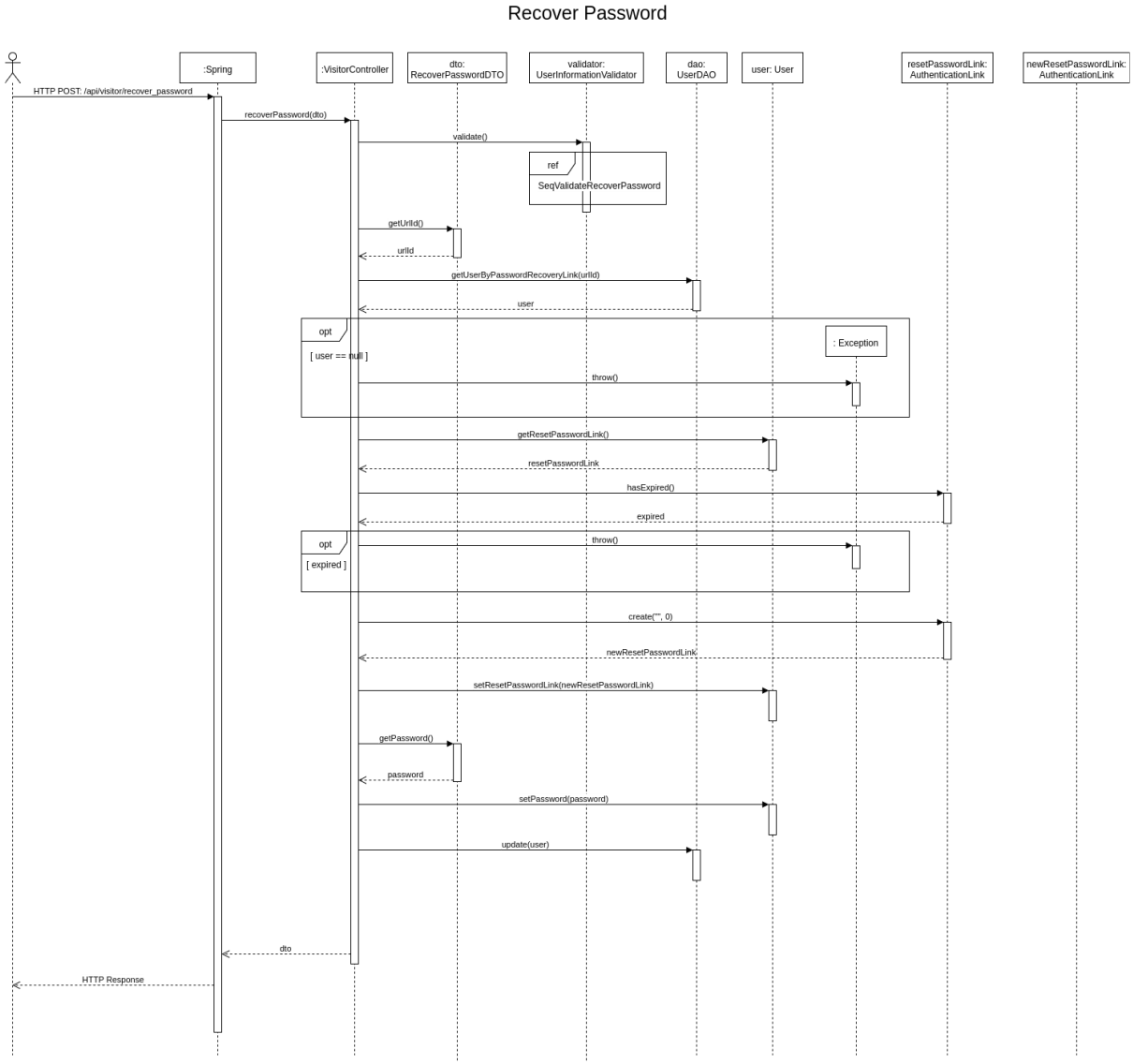
Make Super Admin



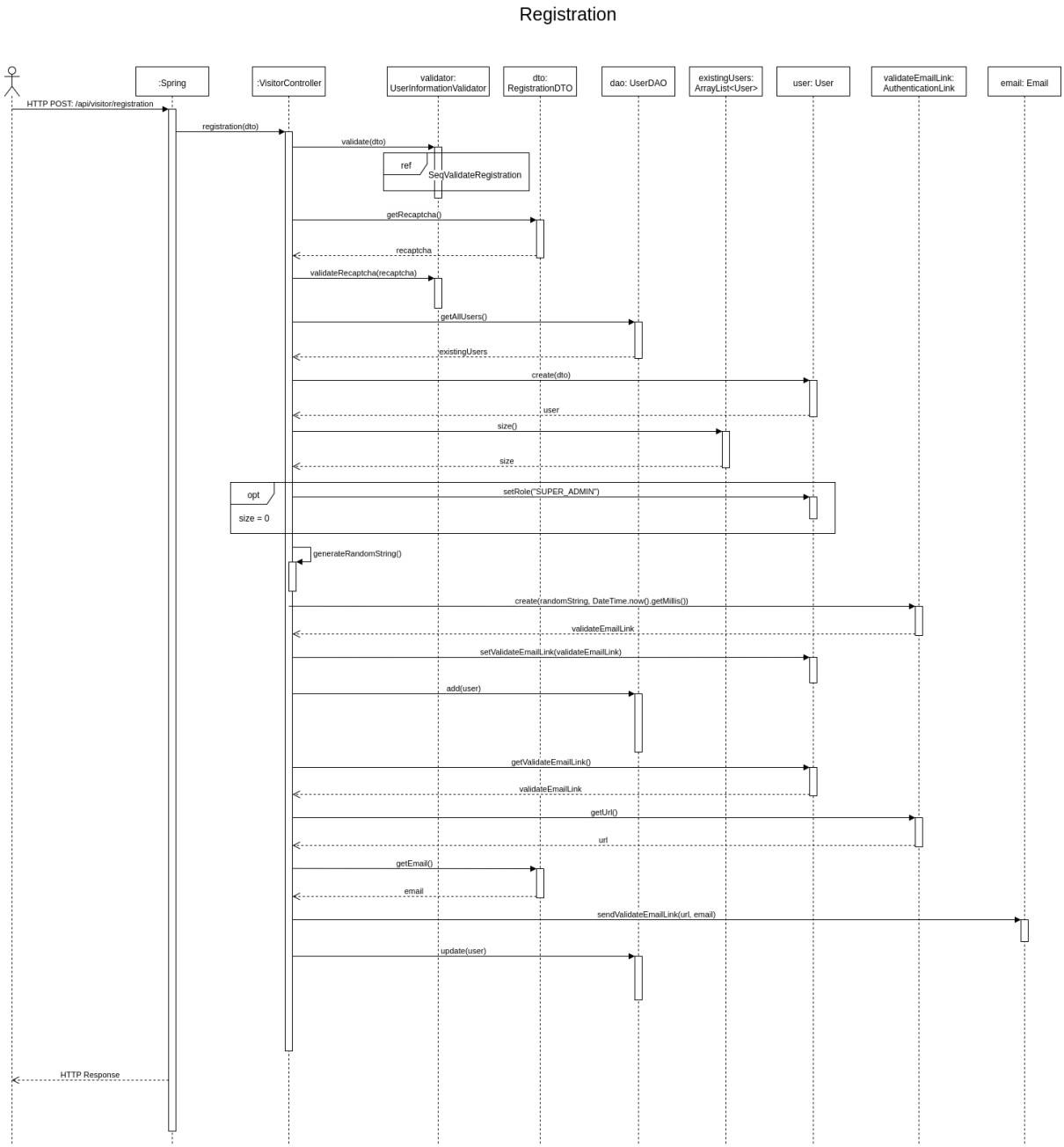
Make User


Make User
:Spring
:AdminController
dao: UserIdDTO
currentUser: CurrentUser
actor: User
target: User
dto: UserIdDTO
HTTP POST: /api/admin/make_user
makeUser(dto)
getEmailAddress()
email
getUserByEmail(email)
create()
actor
actor
getId()
id
getUserById(id)
target
canChangeRoleTo(target, "USER")
ref
Seq_Can_Change_Role_To
canChangeRole
opt
[canChangeRole]
setRole("USER")
update(target)
HTTP Response

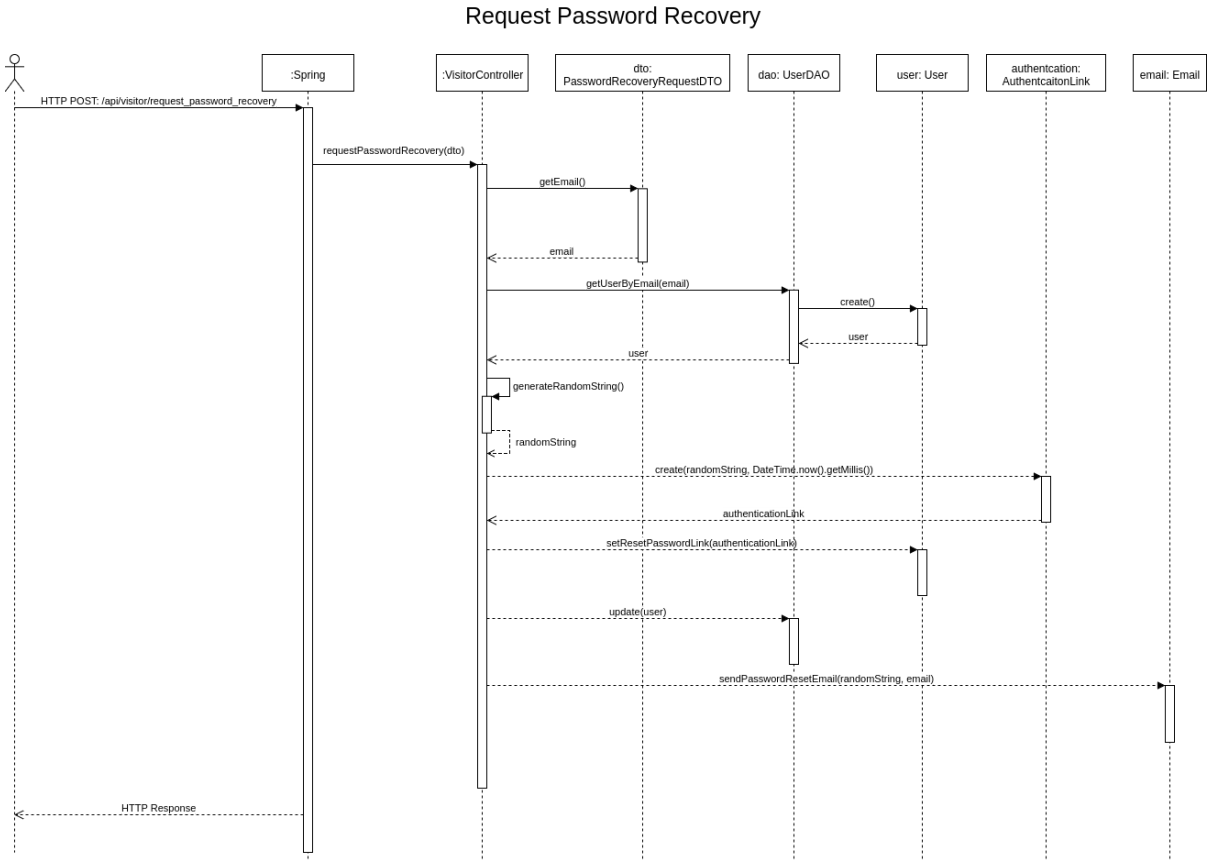
Recover Password



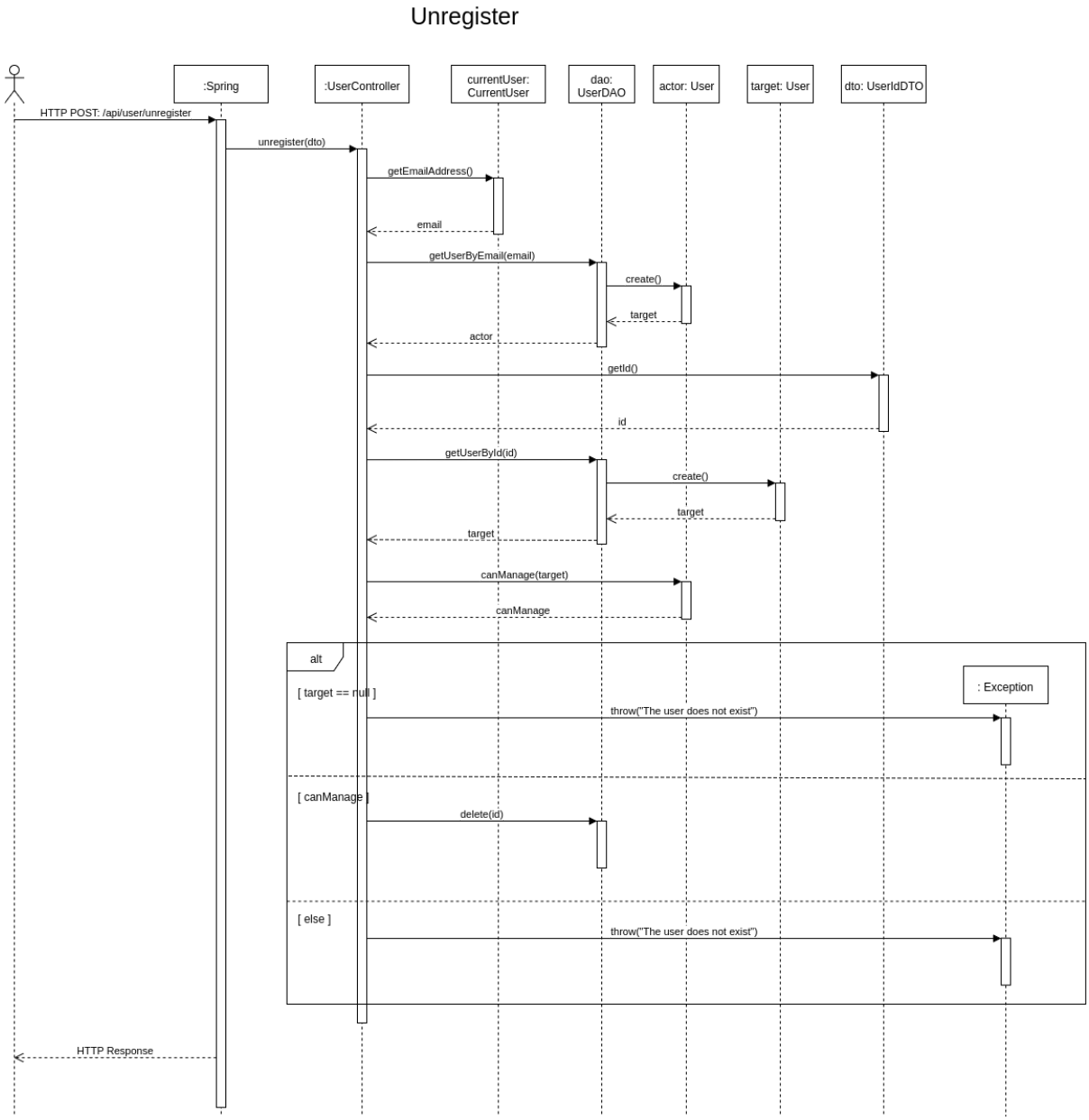
Registration



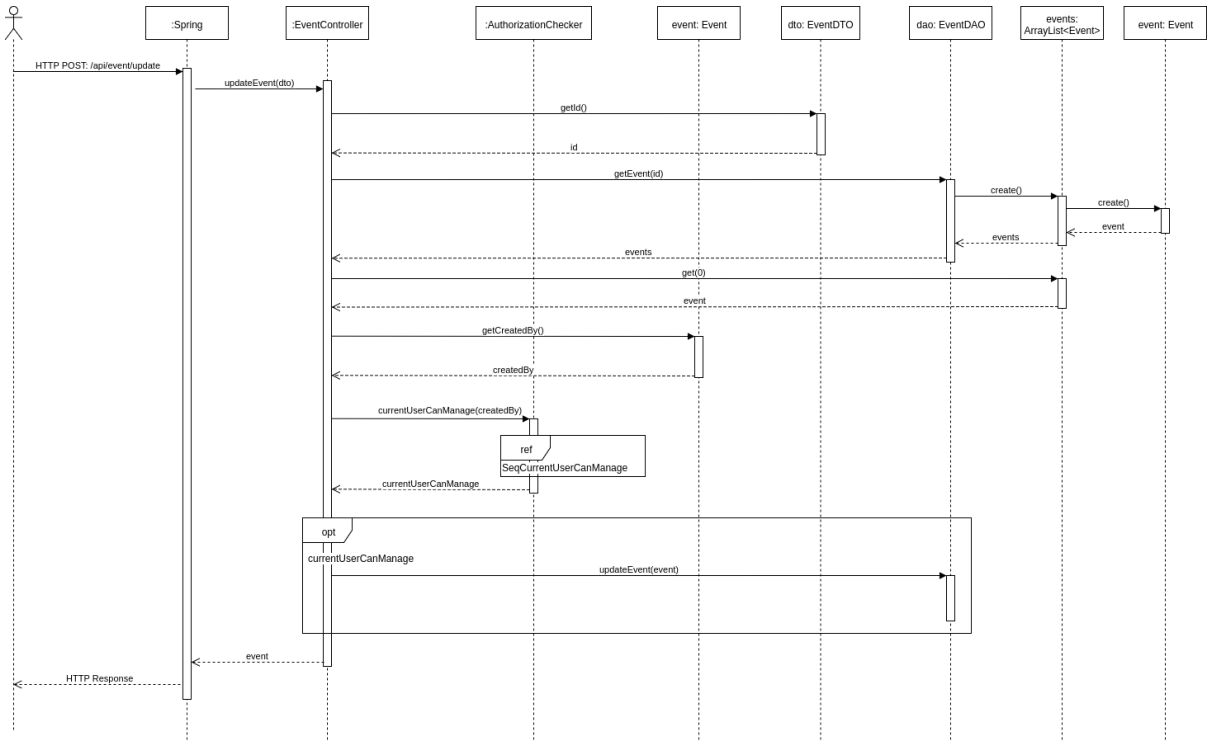
Request Password Recovery



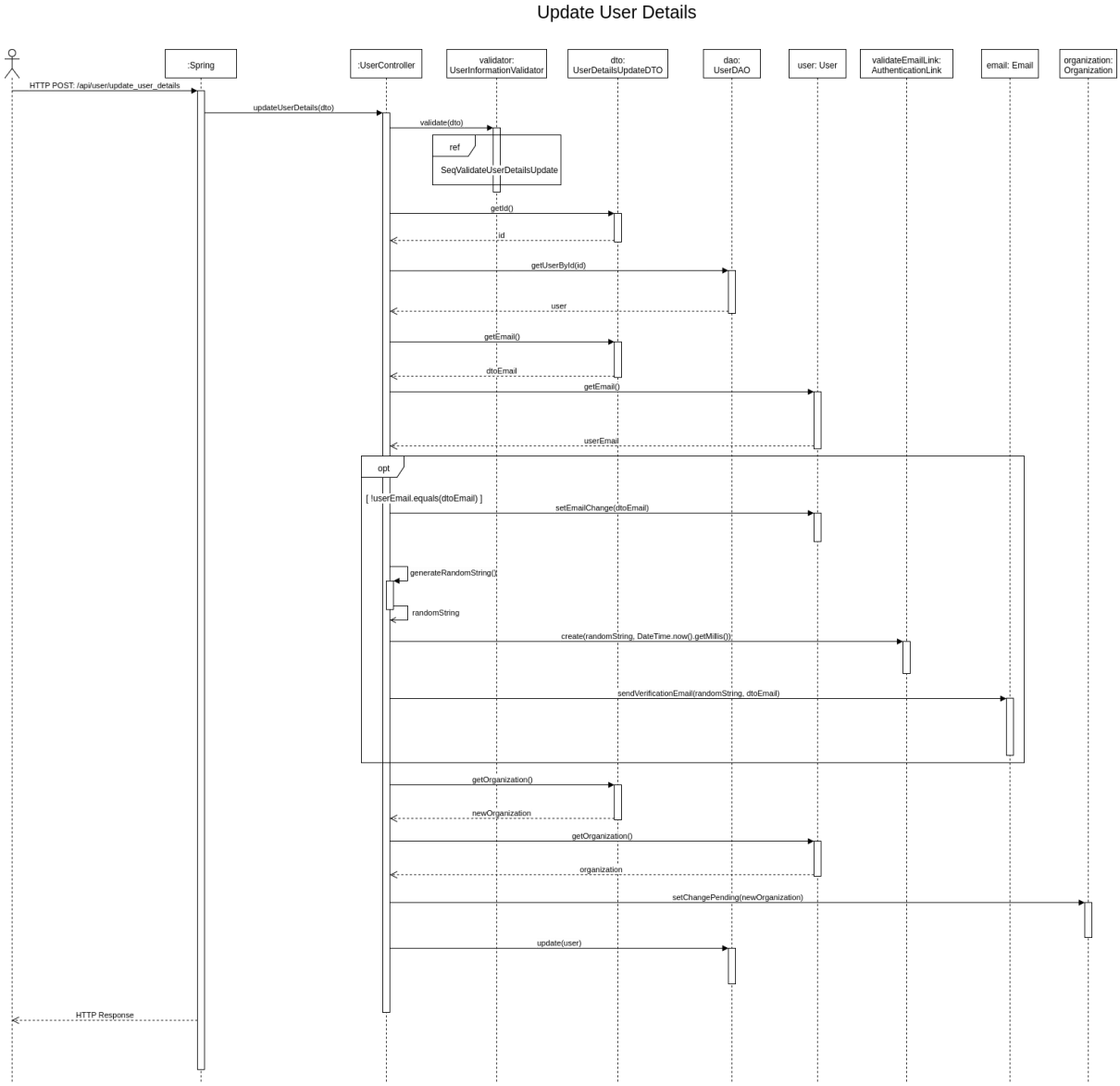
Unregister


Unregister
:Spring
:UserController
currentUser: CurrentUser
dao: UserDAO
actor: User
target: User
dto: UserIdDTO
HTTP POST: /api/user/unregister
unregister(dto)
getEmailAddress()
email
getUserByEmail(email)
create()
target
actor
getId()
id
getUserById(id)
create()
target
target
canManage(target)
canManage
alt
[target == null]
: Exception
throw("The user does not exist")
[canManage]
delete(id)
[else]
throw("The user does not exist")
HTTP Response

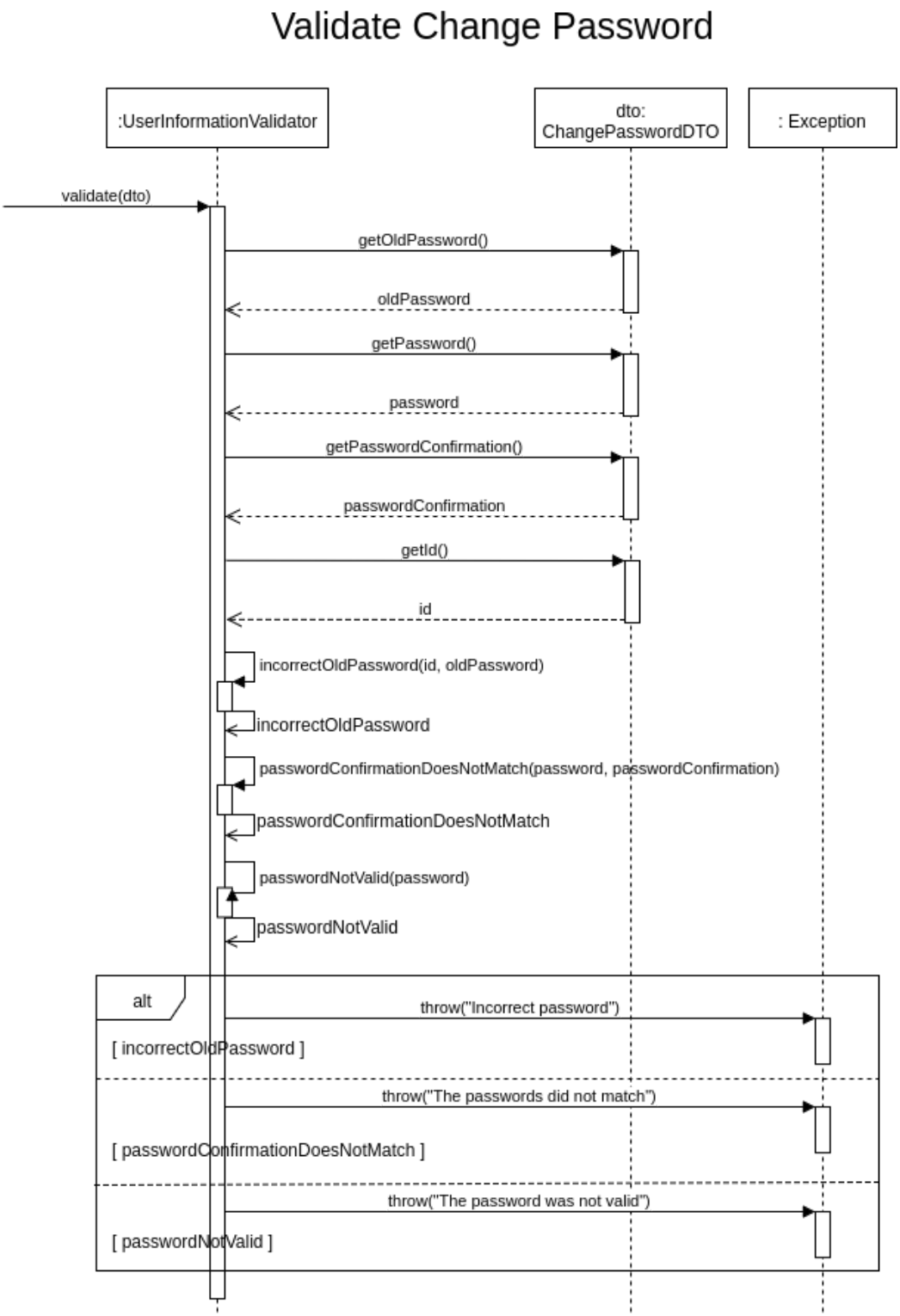
Update Event



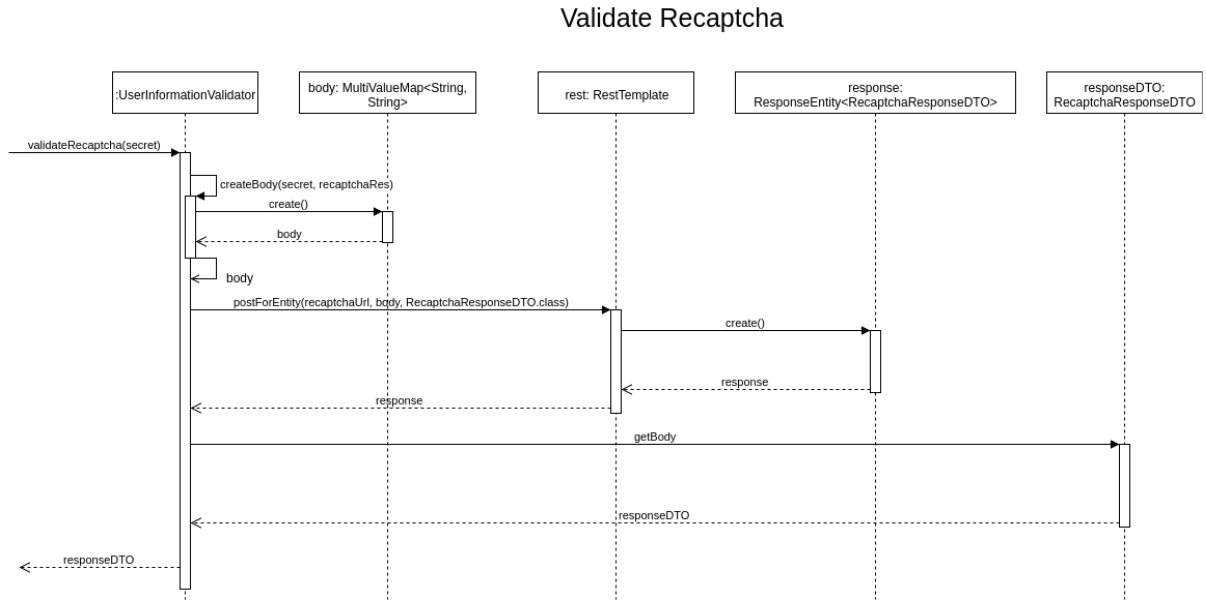
Update User Details



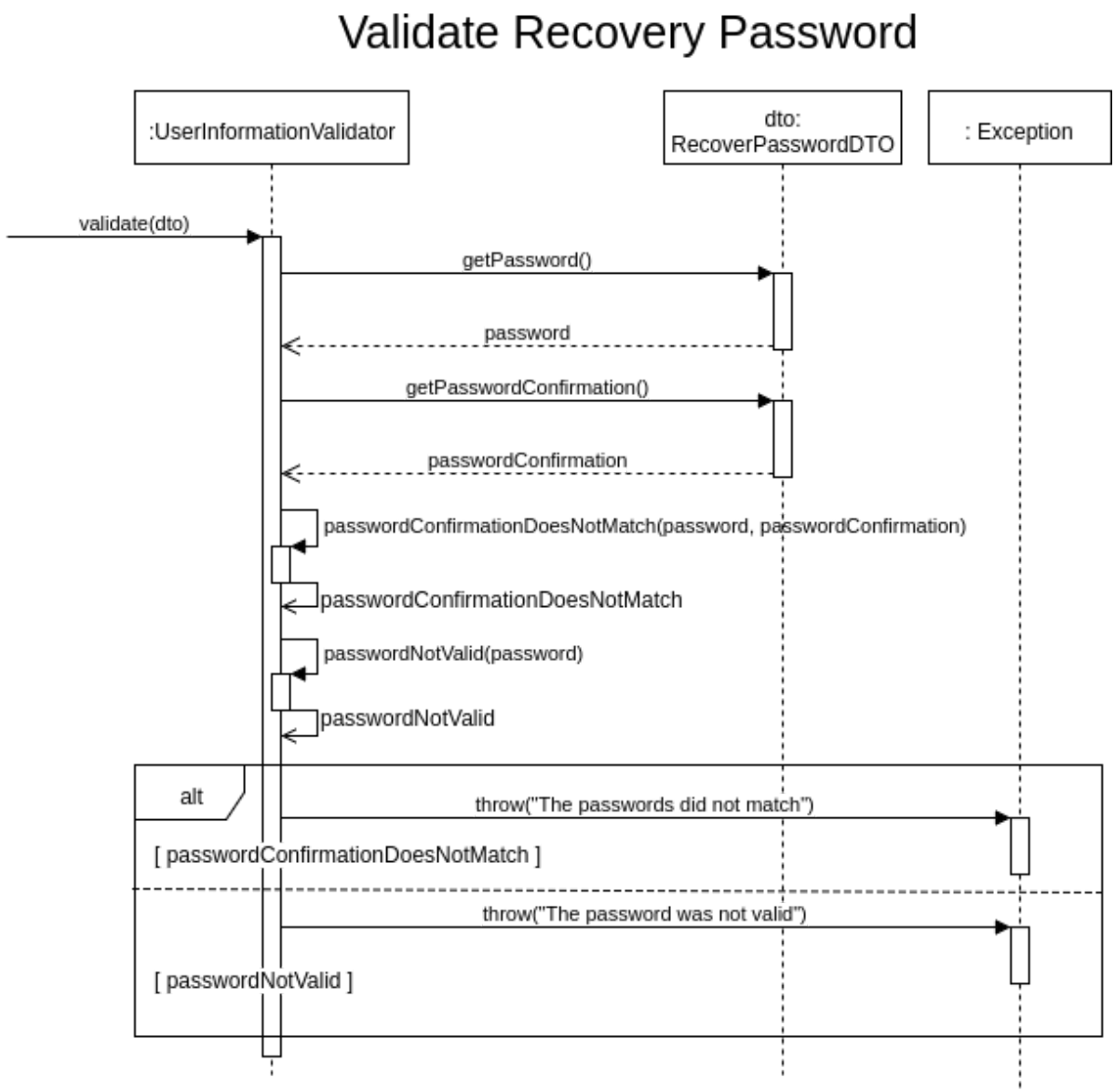
Validate Change Password



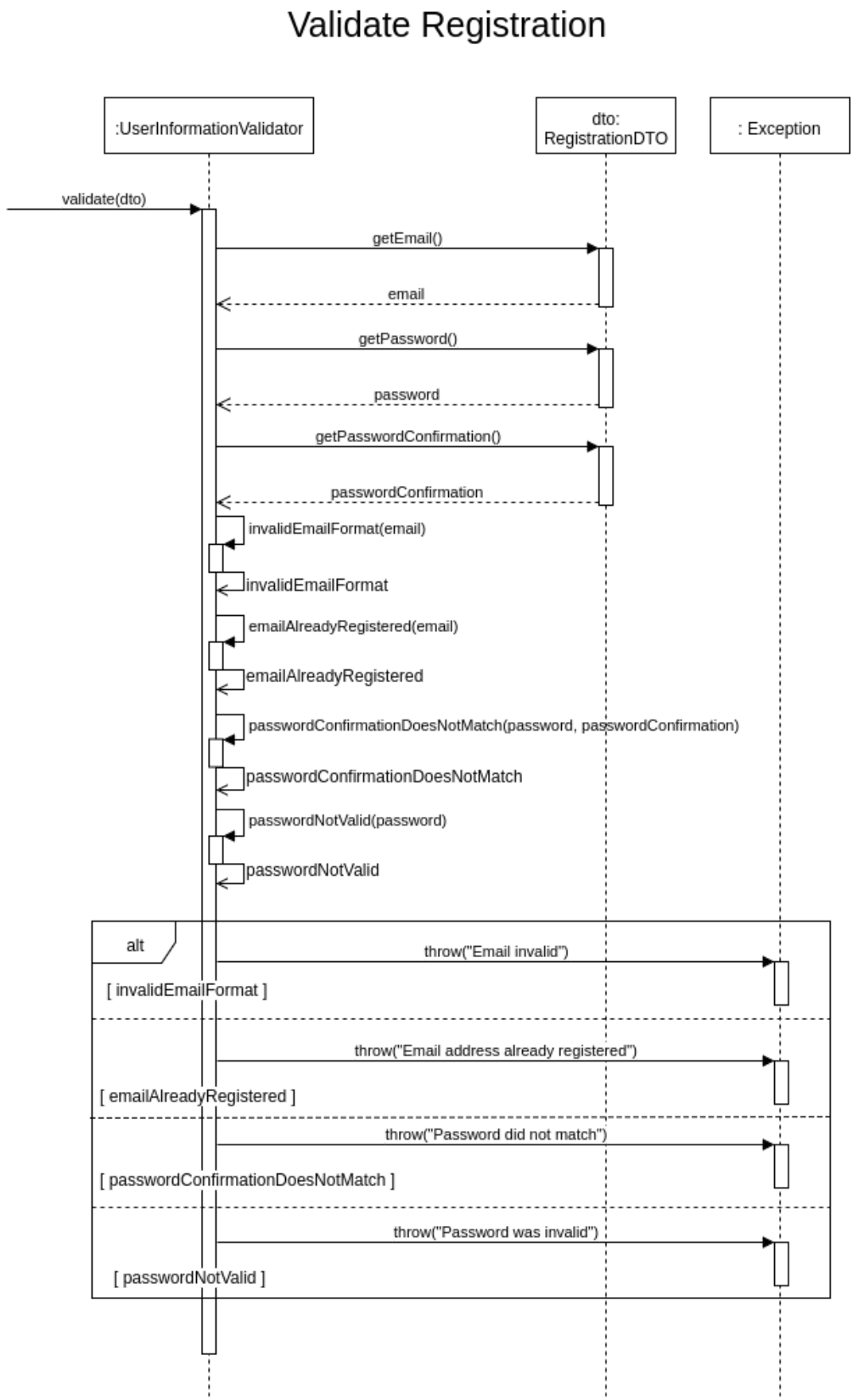
Validate Recaptcha



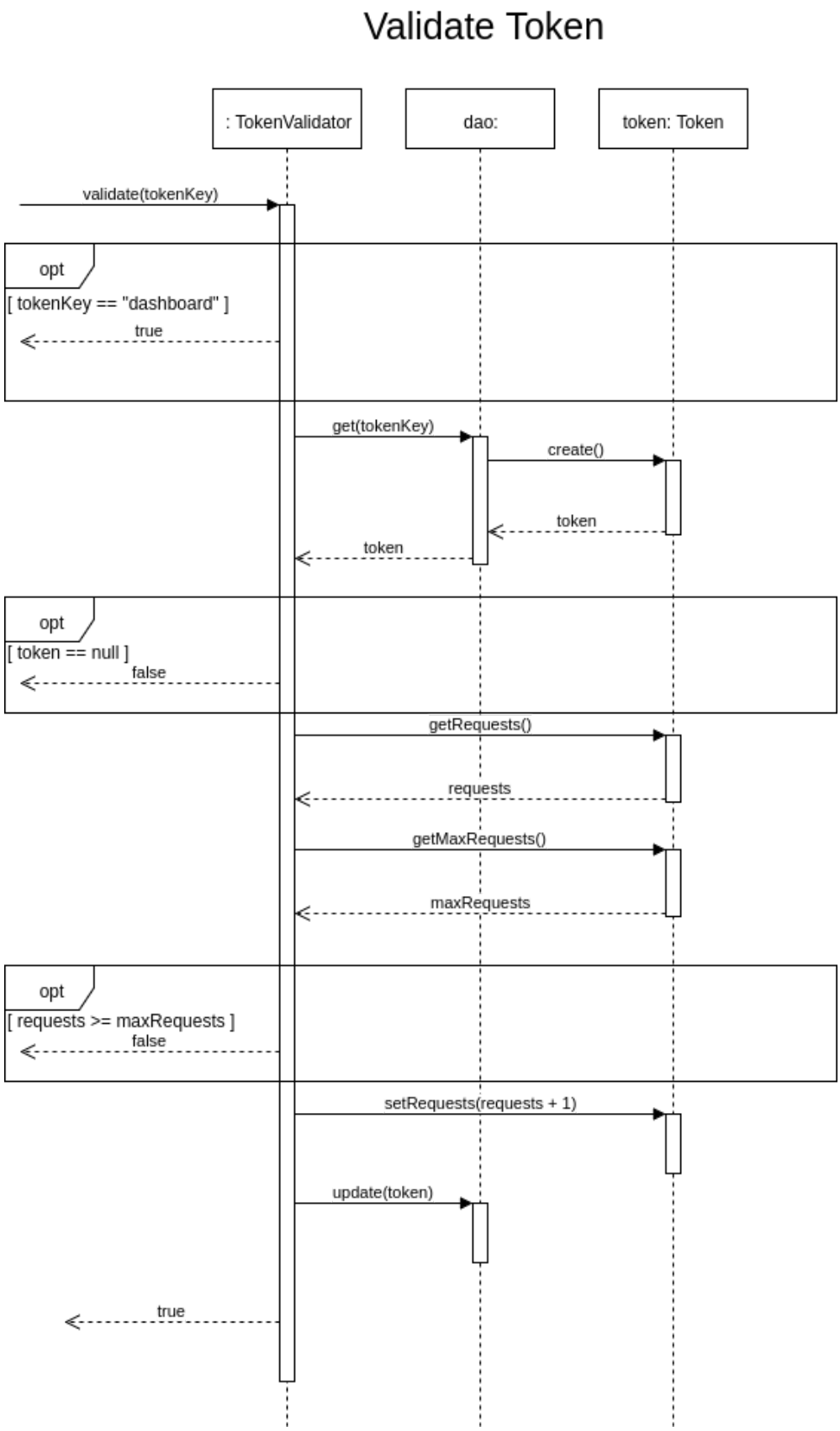
Validate Recover Password



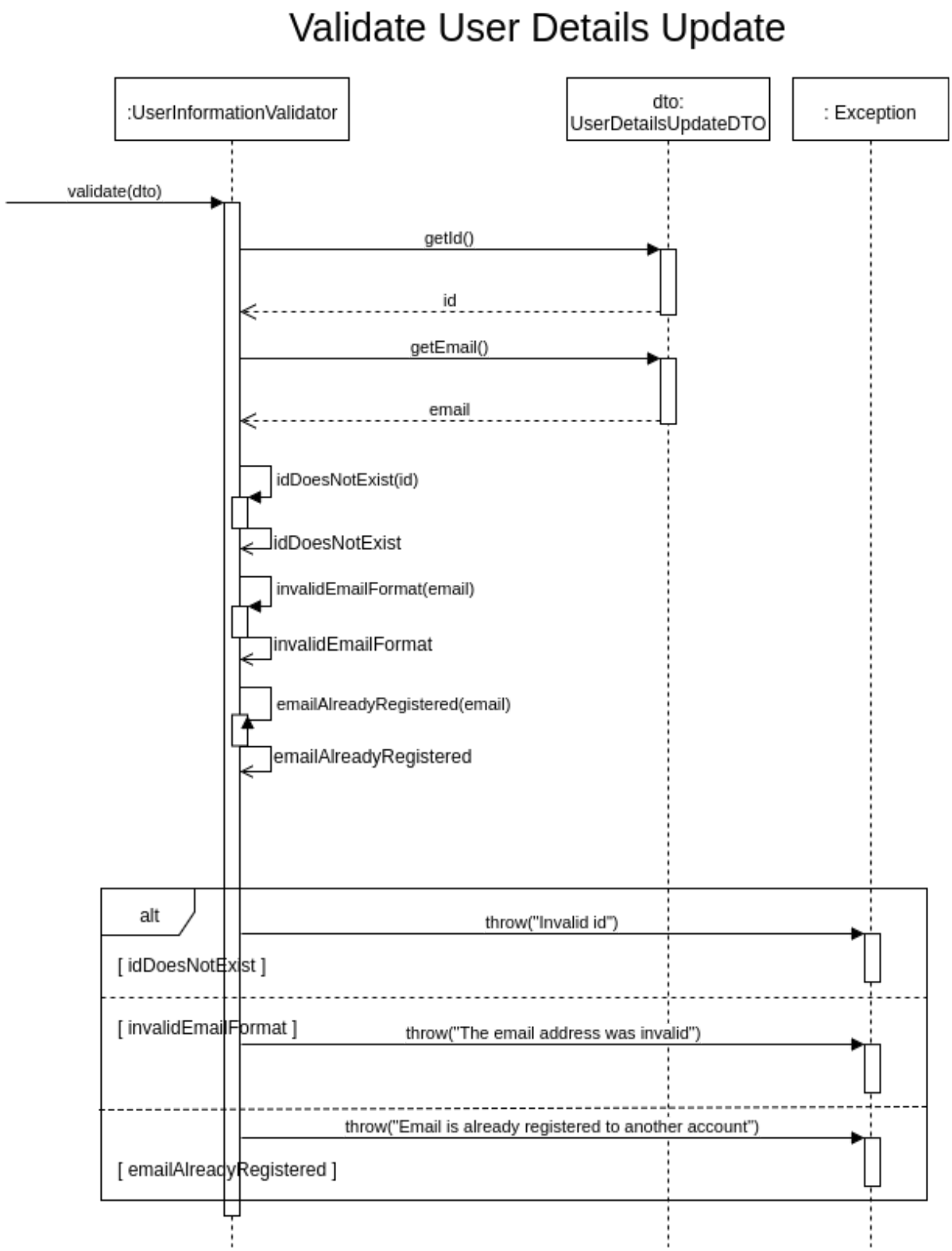
Validate Registration



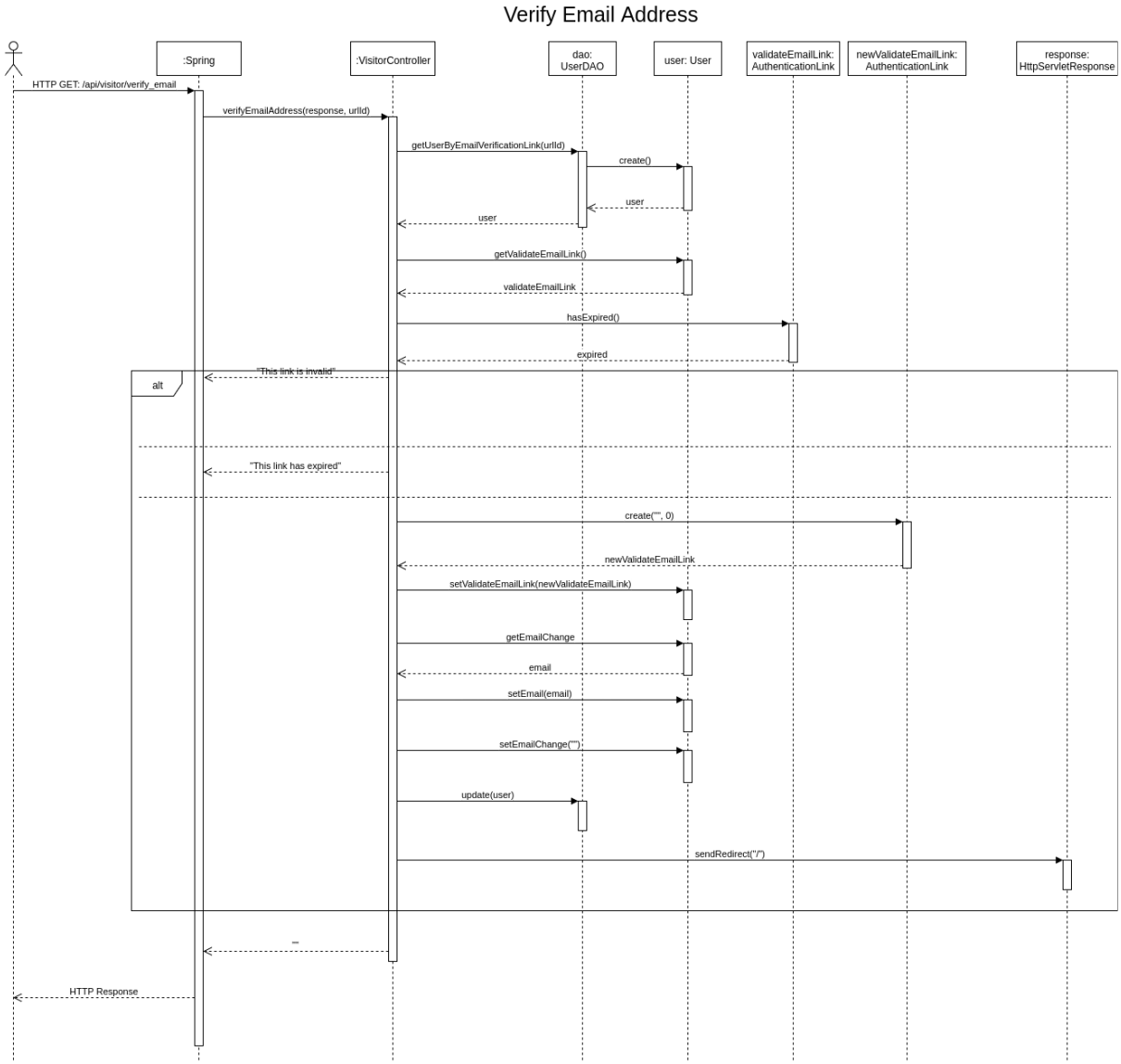
Validate Token



Validate User Details Update

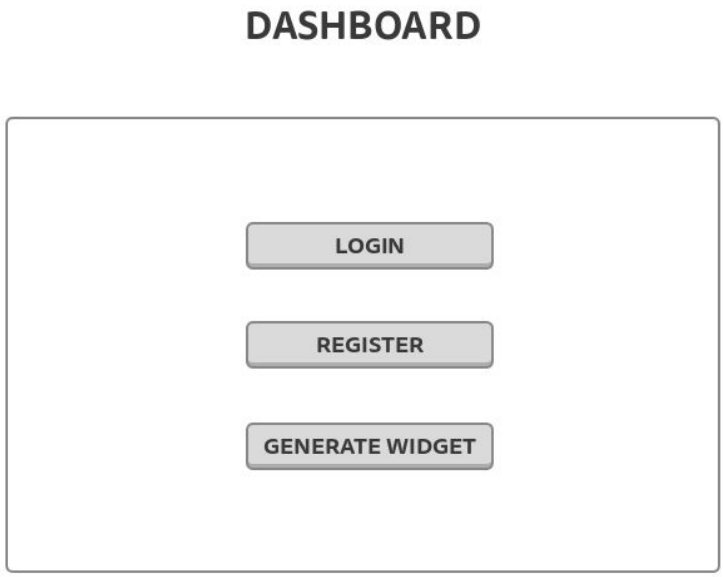


Verify Email Address

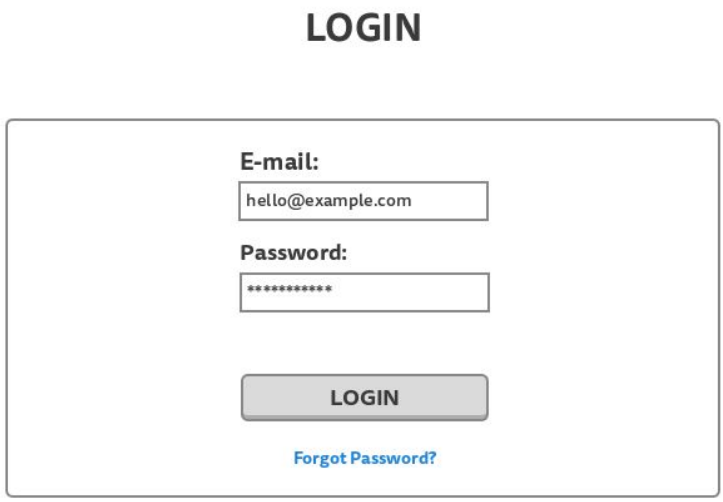


Appendix III: User Interface Wireframes

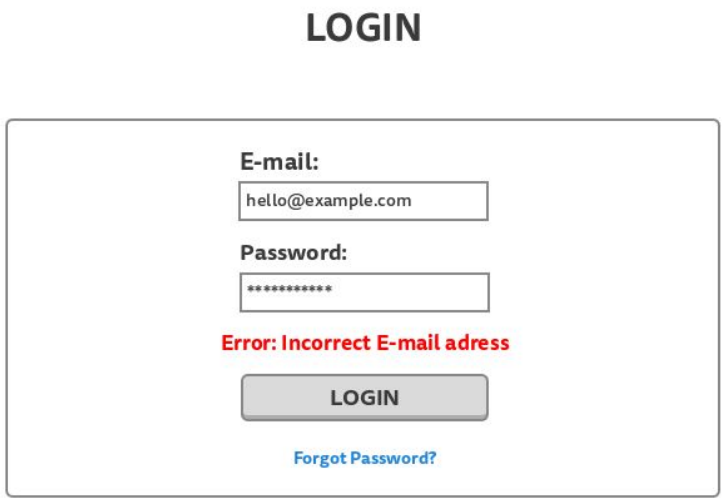
Dashboard Main View



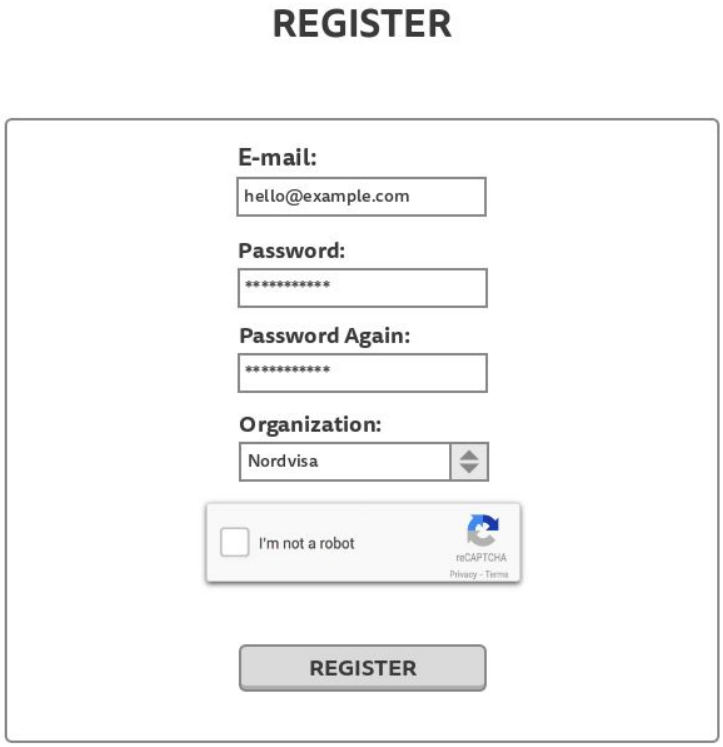
Dashboard Login View



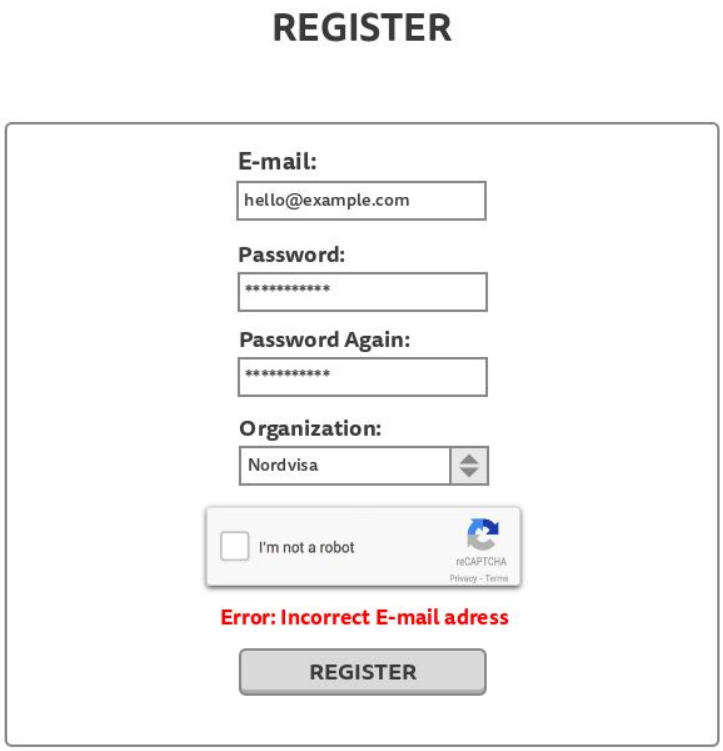
Dashboard Login View - With Error



Dashboard Register View



Dashboard Register View - With Error

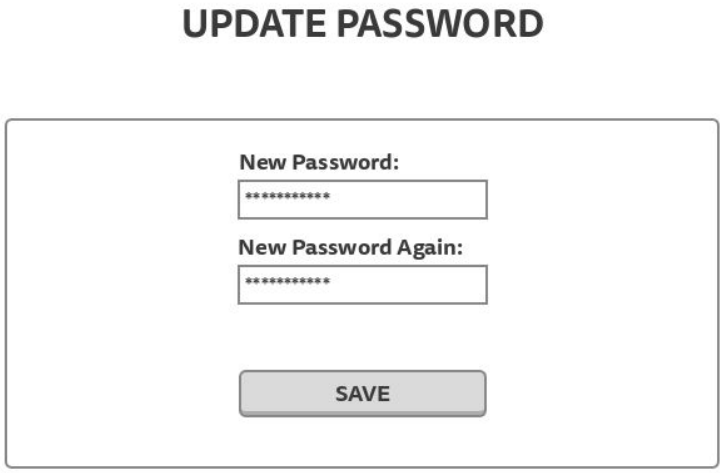


Dashboard Register View - New Organization

Dashboard Recover Password View

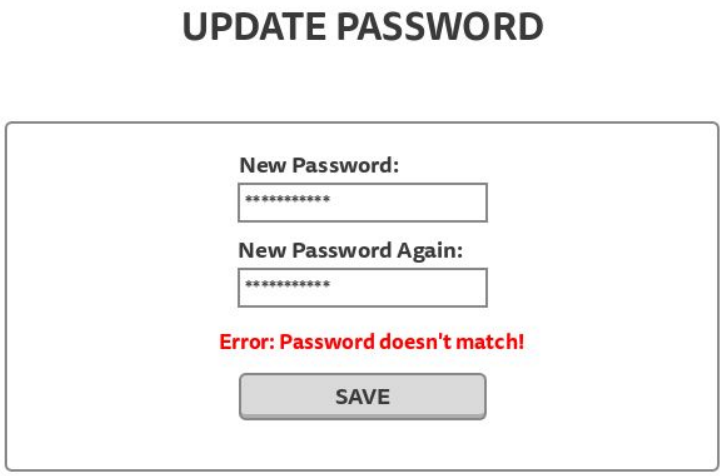
Dashboard Recover Password View - With Error

Dashboard New Password View

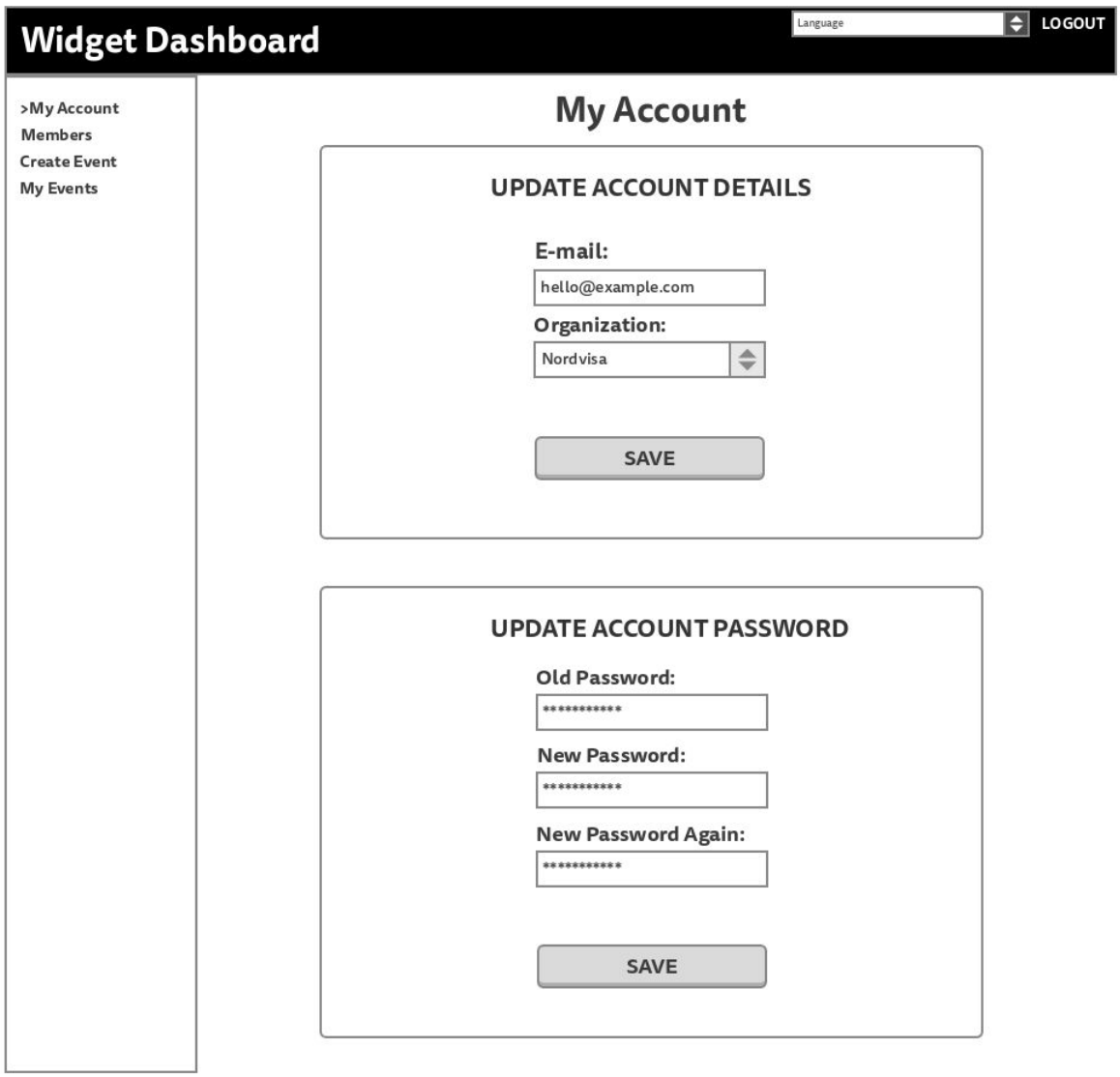


Dashboard New Password View - With Error

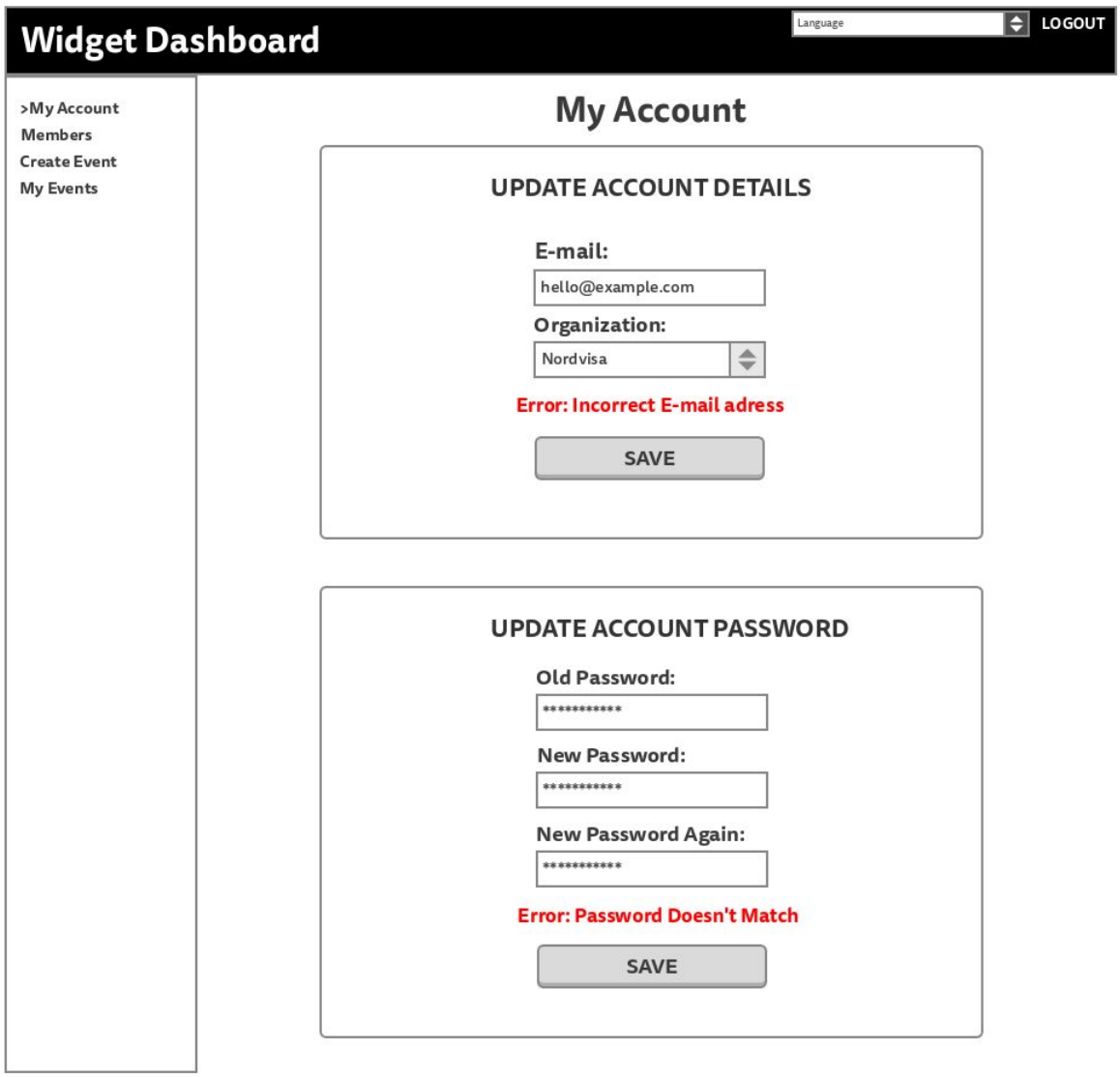
’



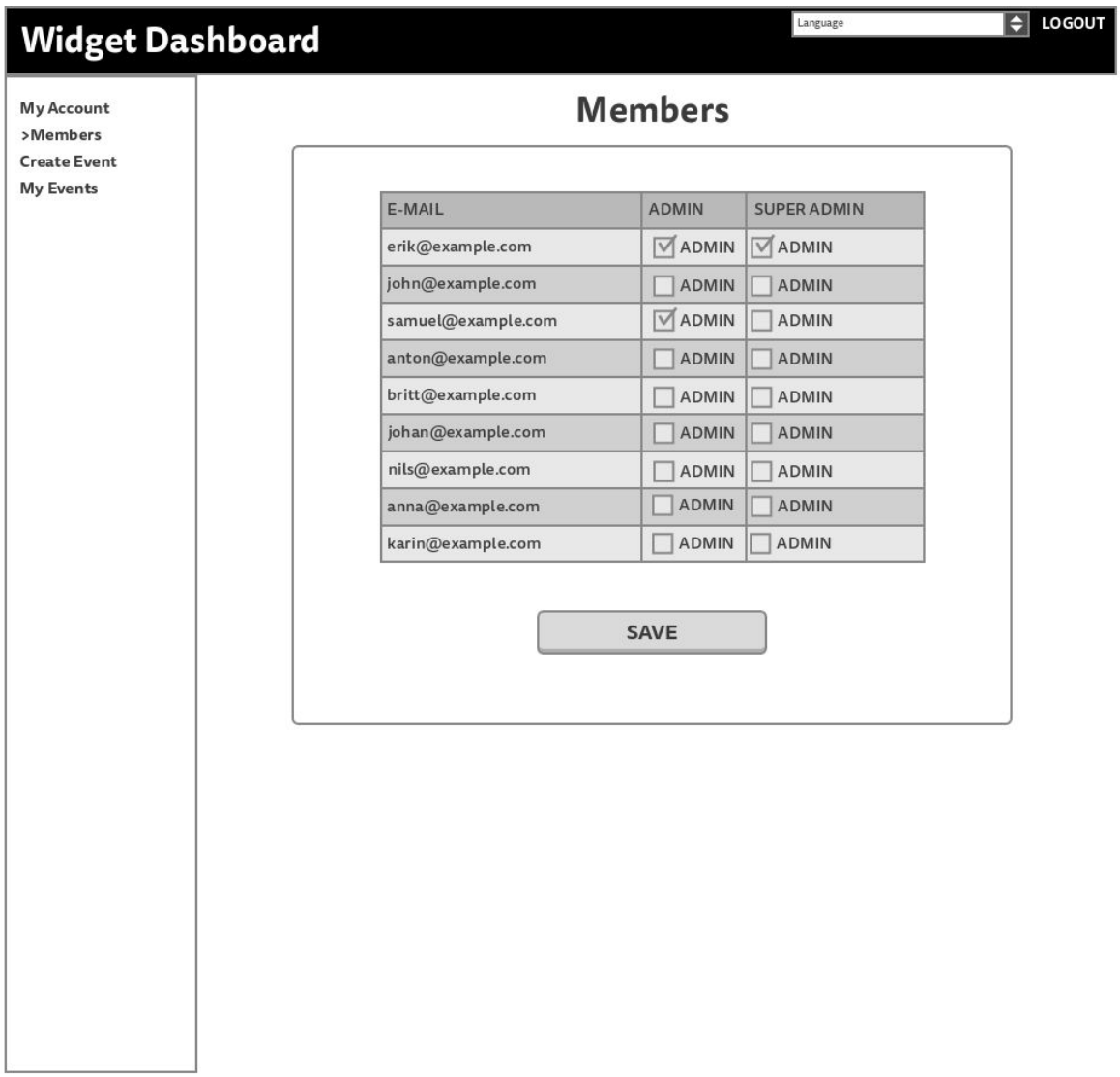
Dashboard My Account Page View



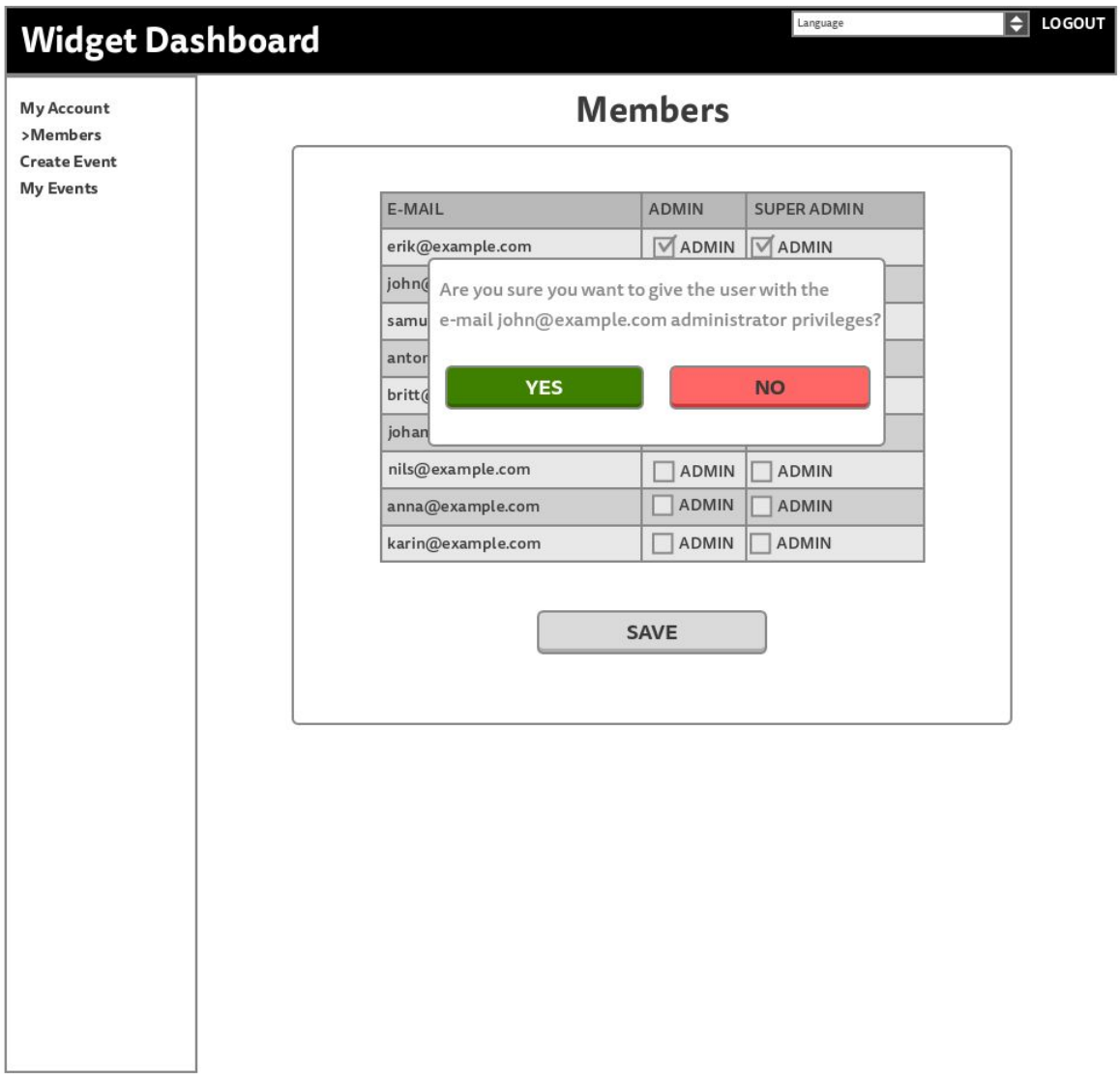
Dashboard My Account Page View - With Error



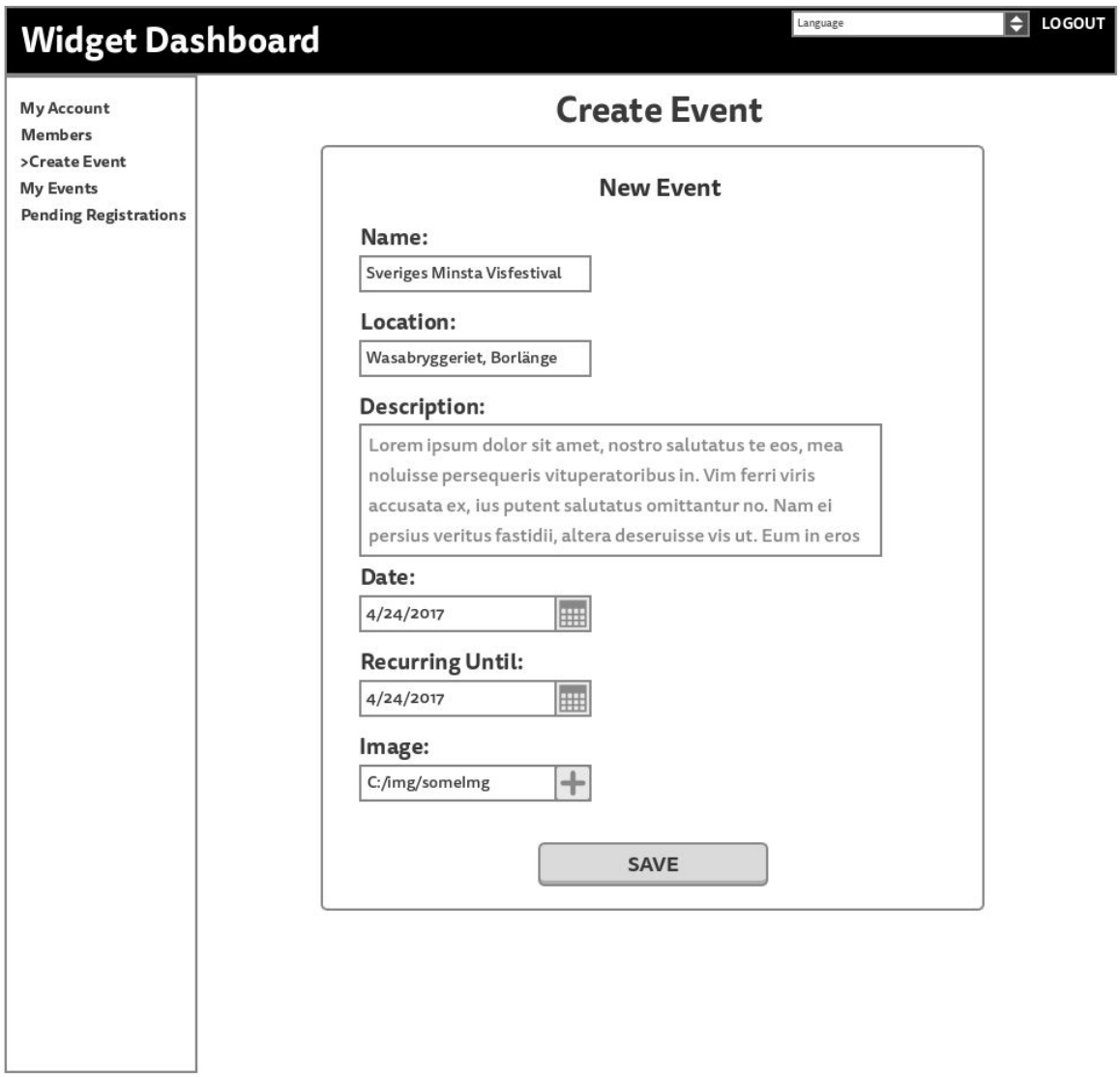
Dashboard Members Page View



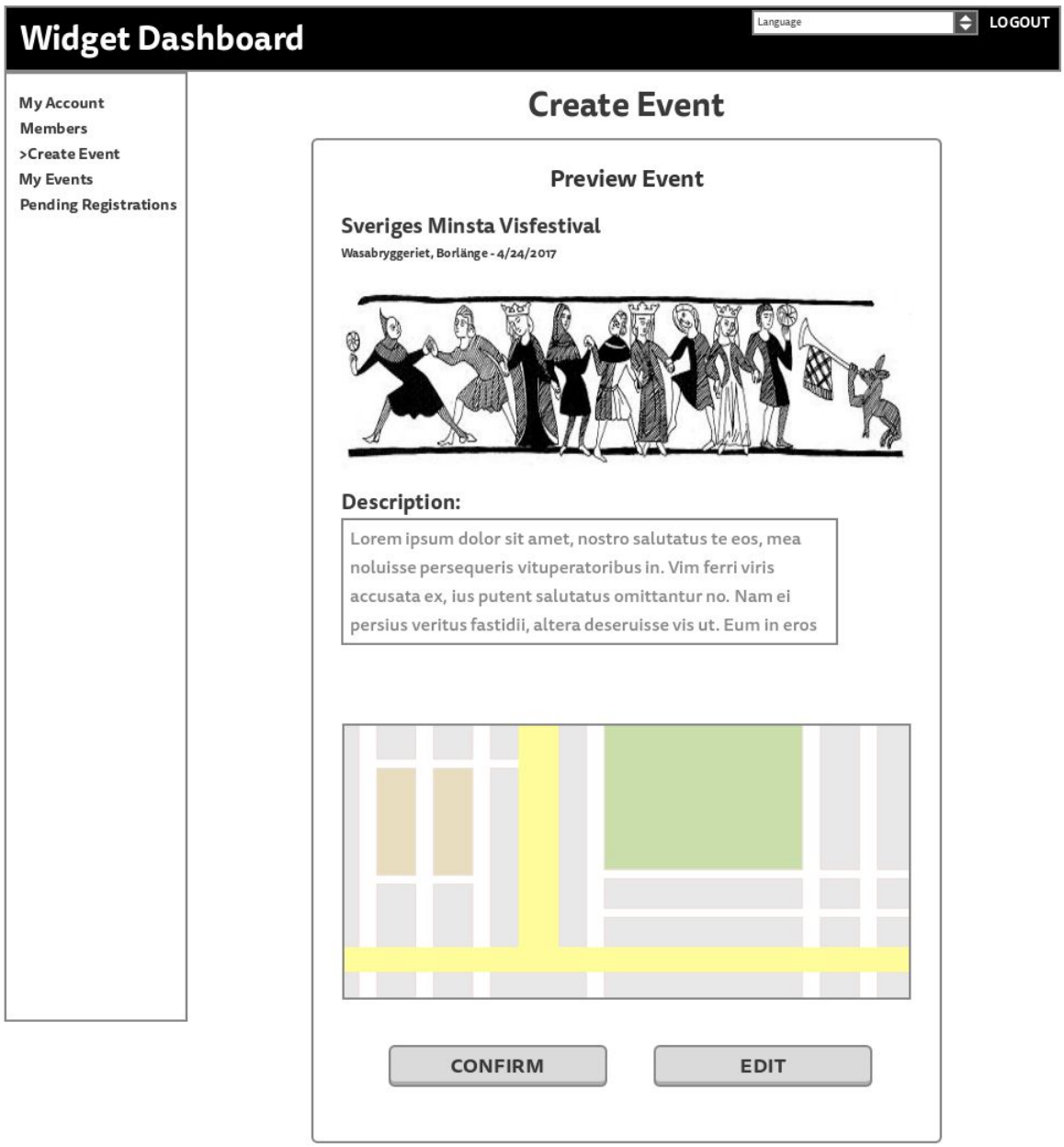
Dashboard Members Page View - With PopUp



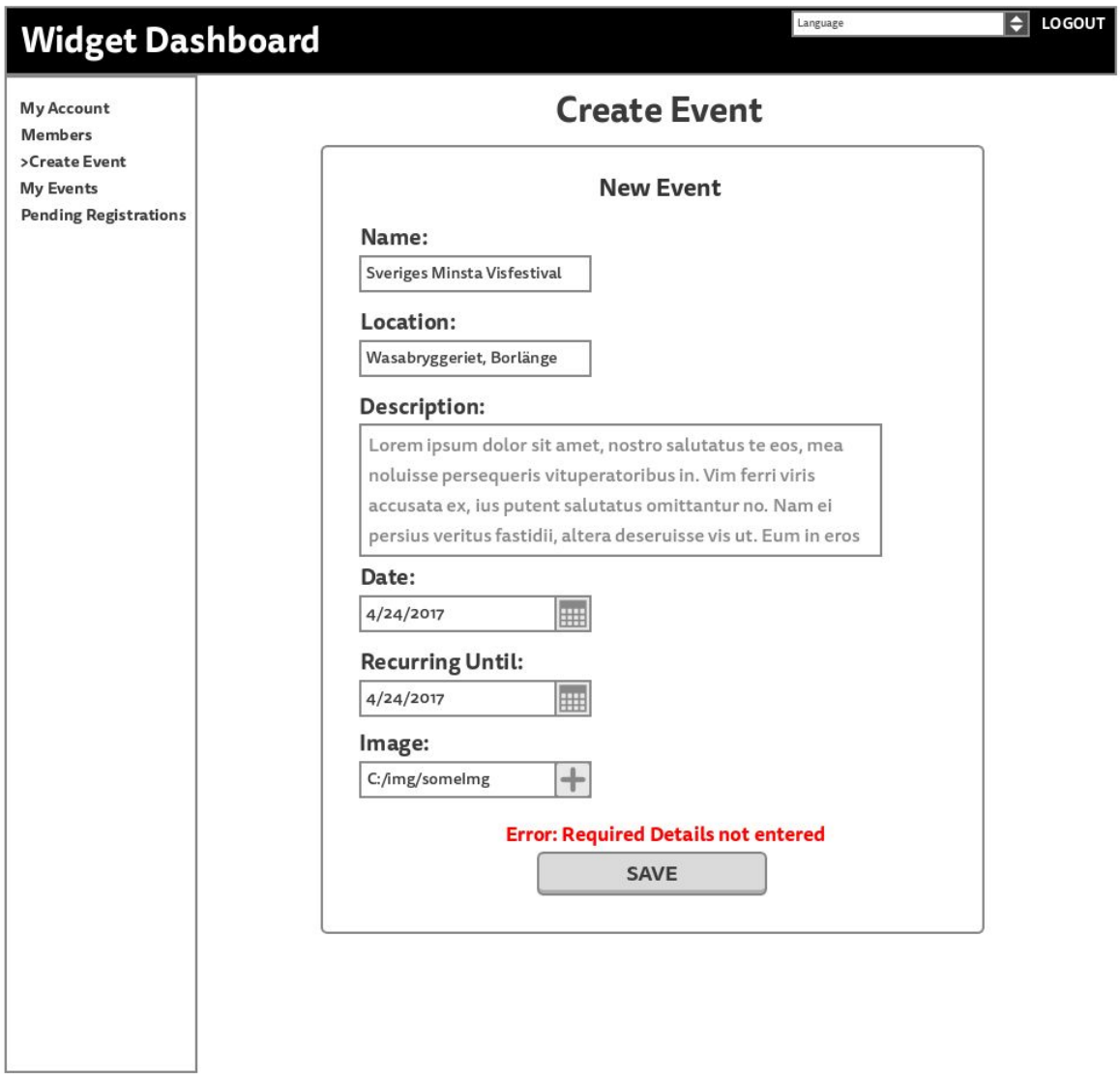
Dashboard Create Event Page View



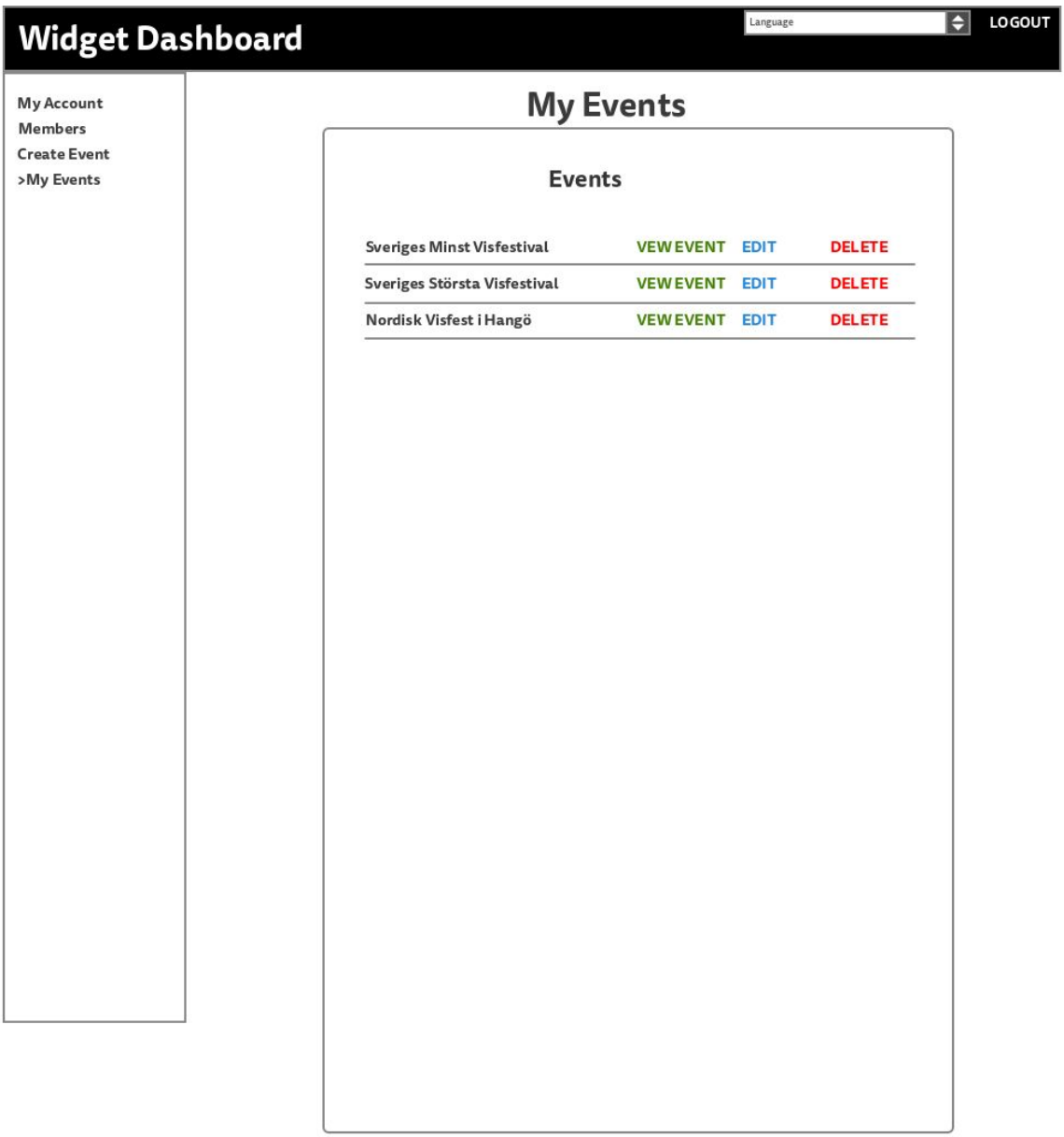
Dashboard Preview Event Page View



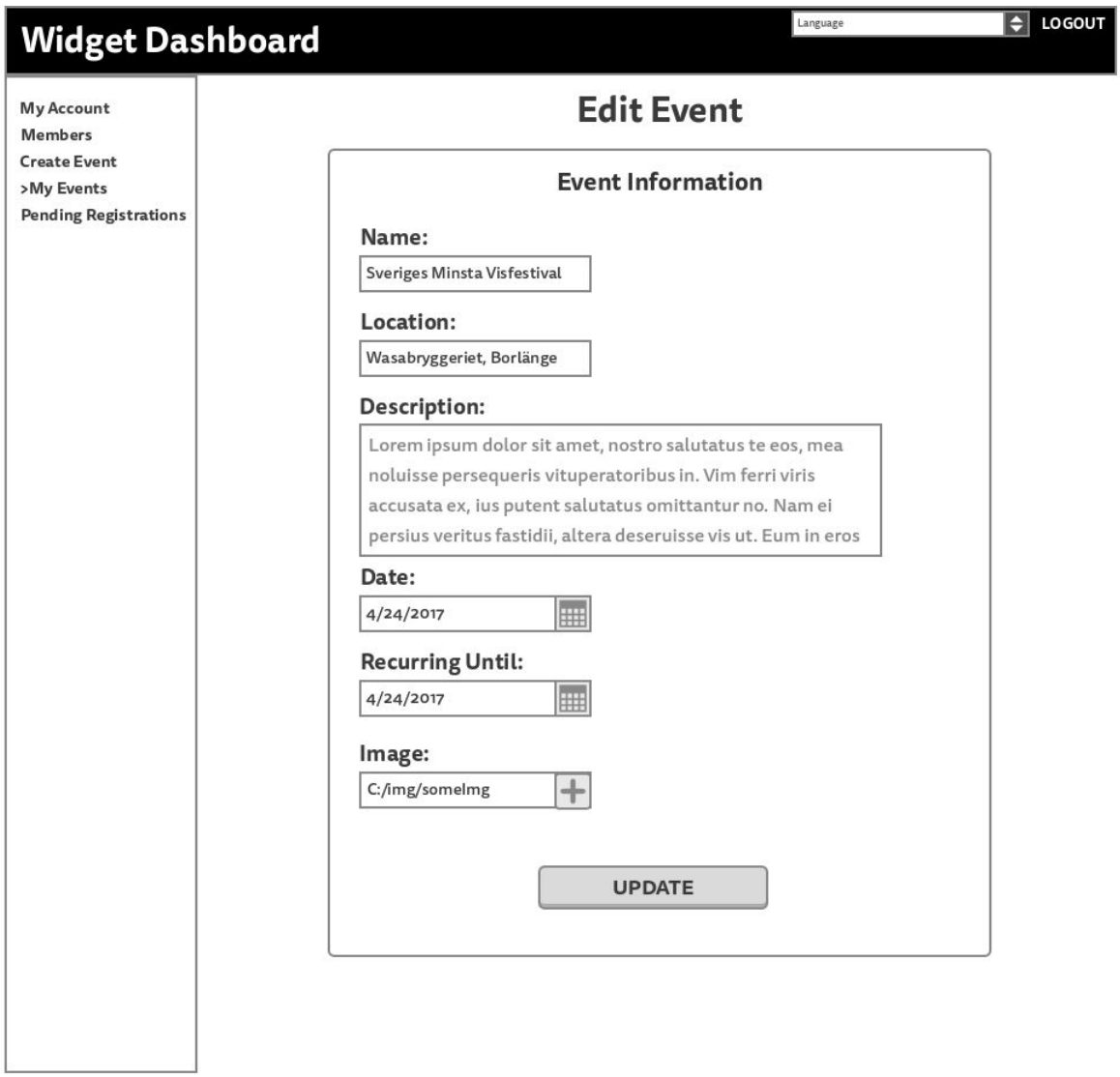
Dashboard Create Event Page View - With Error



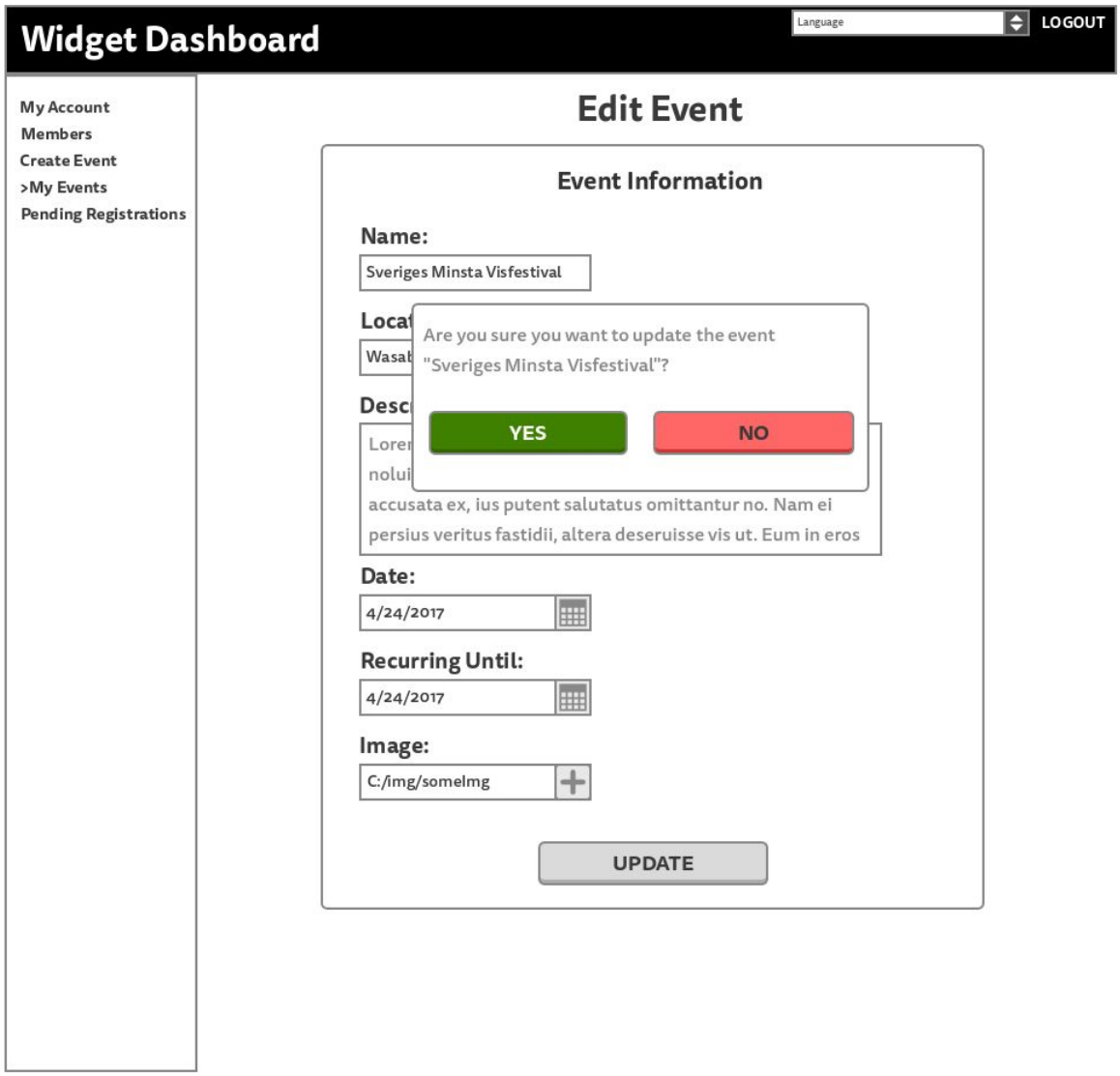
Dashboard My Events Page View



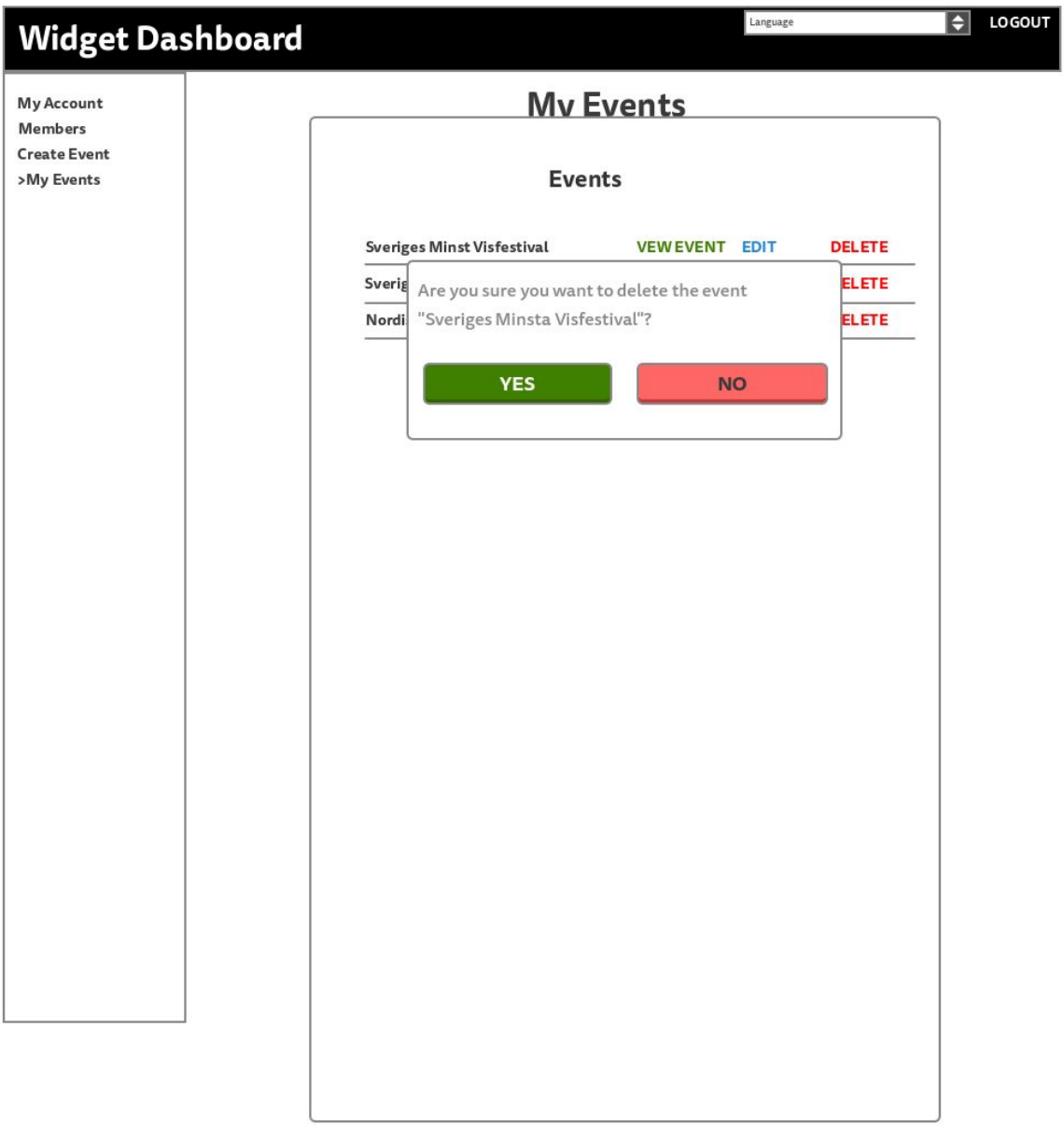
Dashboard Edit Event Page View



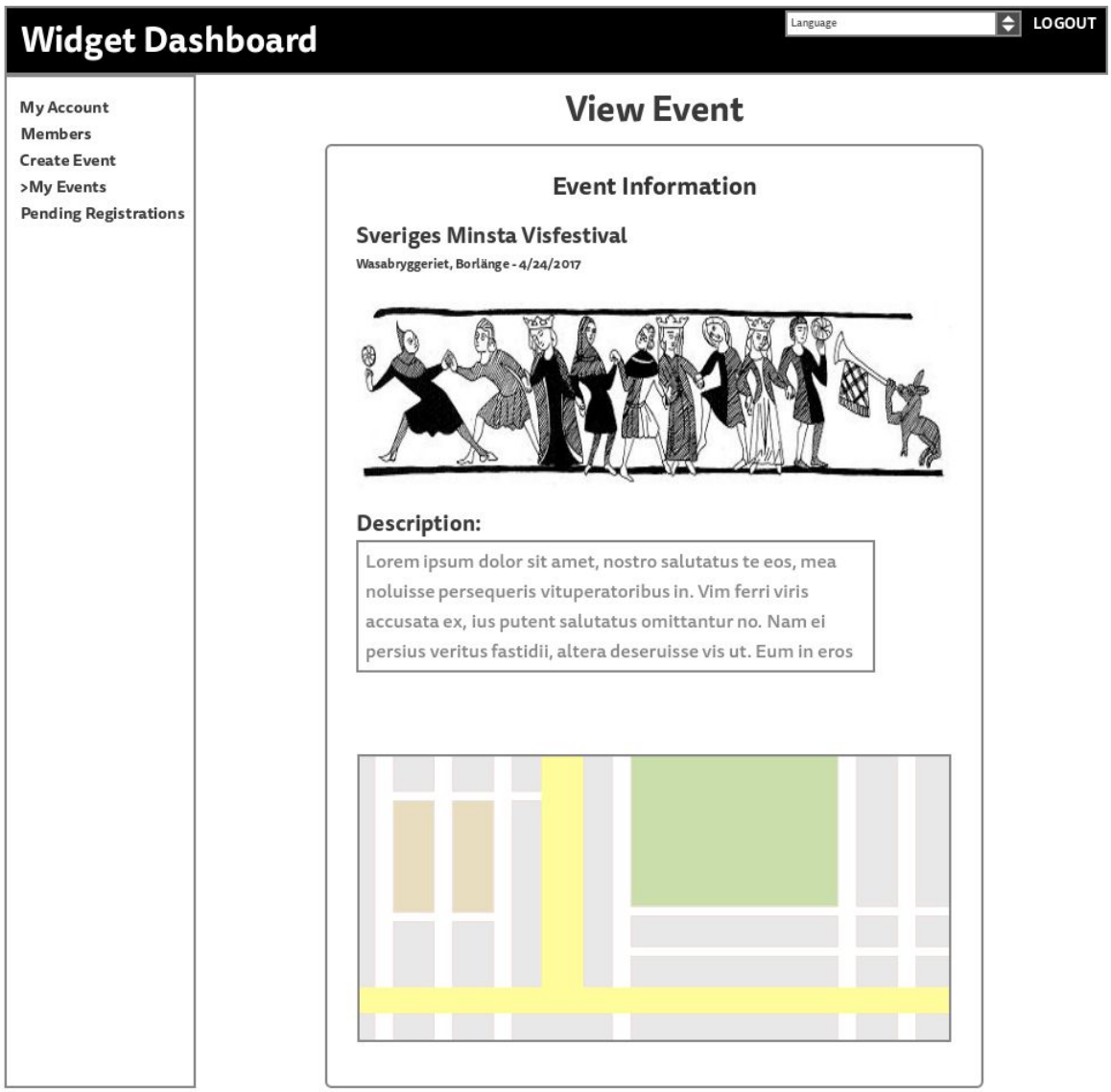
Dashboard Edit Event Page View - With PopUp



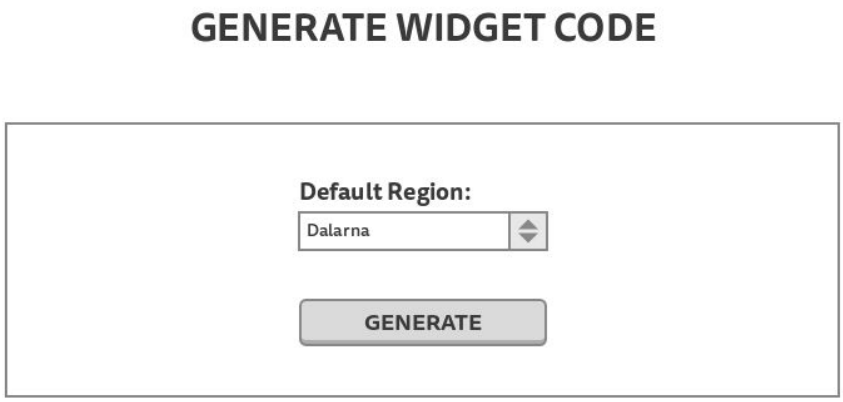
Dashboard Delete Event View



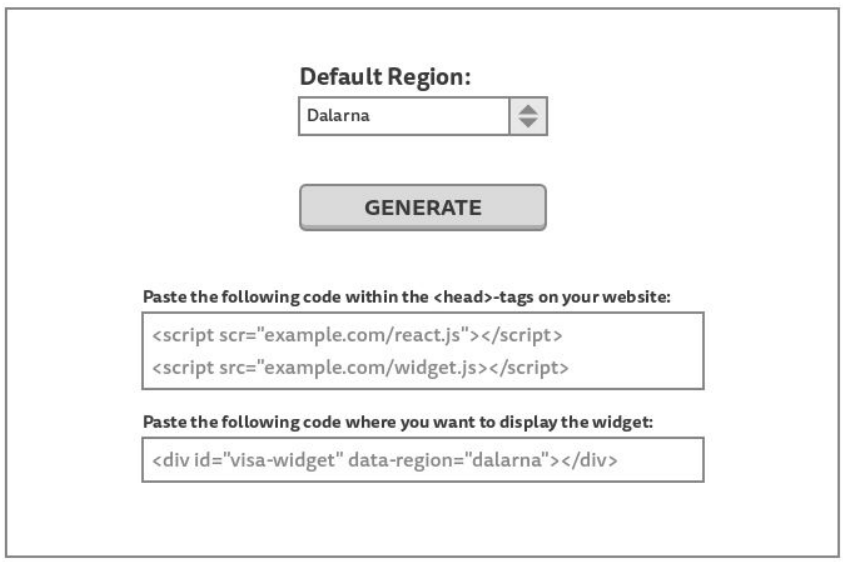
Dashboard View Event Page View



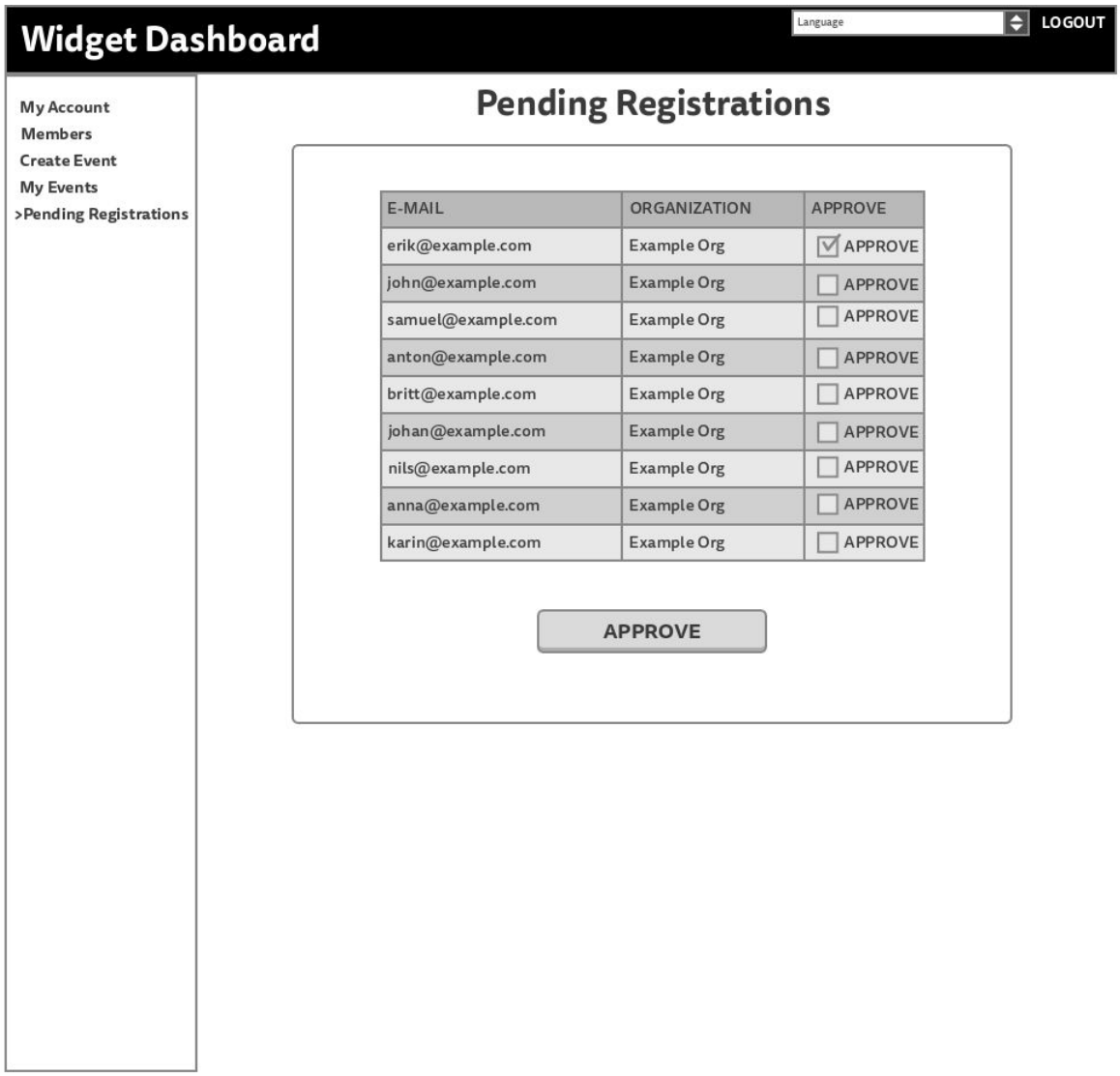
Dashboard Generate Widget Page View



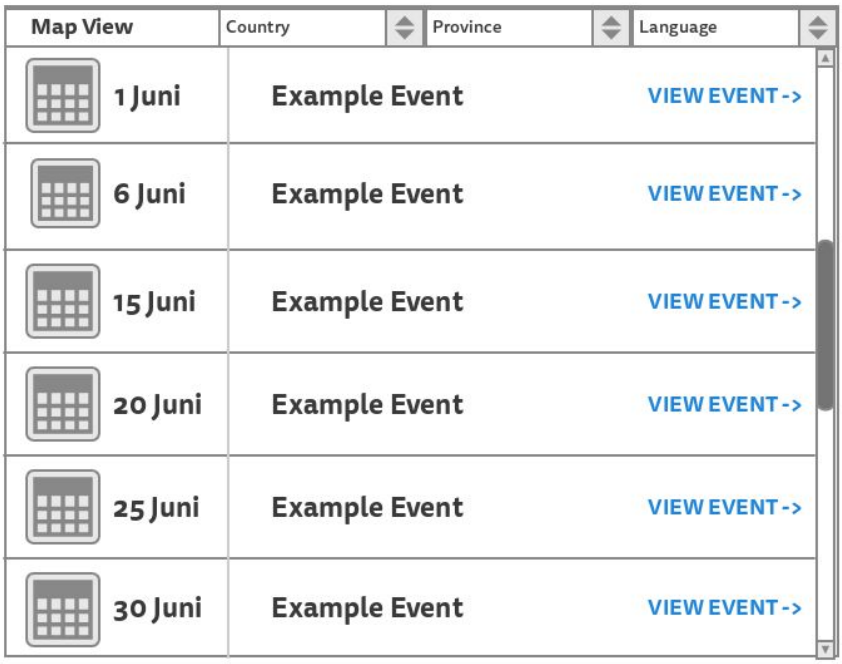
Dashboard Generate Widget Page View - Generated



Dashboard Pending Registrations Page View



Widget Calendar View



Widget Map View

Widget Single Event View

Calendar View	Country	Province	Language

Sveriges Minsta Visfestival

Wasabryggeriet, Borlänge - 4/24/2017

Description:

Lorem ipsum dolor sit amet, nostro salutatus te eos, mea noluisse persequeris vituperatoribus in. Vim ferri viris accusata ex, ius putent salutatus omittantur no. Nam ei persius veritus fastidii, altera deseruisse vis ut. Eum in eros