Security Engineering Project

Hoàng Nguyễn David Basin Srđan Krstić
{hoang.nguyen,basin,srdan.krstic}@inf.ethz.ch

In this project you are tasked with designing and implementing a secure web application, called Event Platform twice: with the νActionGUI tool and with (Python and) Flask. The project consists of three parts. In the first two pages, this document describes the overall project structure. Next, it describes the web application's functional (Section 1), security (Section 2), and privacy (Section 3) requirements. Finally, it explains each part of the project (Section 4).

Technical Questions Technical questions can be addressed to Marko Lisicic preferably during the lab sessions (Fridays, 10:15-12:00, CHN D 46) or via the Moodle Project forum.

Clarifications If any part of the project description is unclear, please contact the teaching assistants (TAs) for clarifications, preferably by leaving a comment directly on this project description. Please use only comments (by right-clicking on selected text) and rather than suggestions (by editing the text directly). Please address any solution-specific questions (e.g., including your code snippets) to Hoang Nguyen (hoang.nguyen@inf.ethz.ch).

Logistics

- At the beginning of the project, each student will be granted access to a private repository on ETH's D-INFK Gitlab server (https://gitlab.inf.ethz.ch/). If you haven't received the access to your private repository, please inform the TAs as soon as possible. Use only these repositories for your work, do not share your solution.
- At the end of each project part, you are required to submit your code through a dedicated submission tab on Moodle. Detailed instructions regarding what to submit can be found in the table below and in the description of each part.
- During the project period, if the discussion results in changes to the project description, ~~the old text will be struck through~~ and replaced with new text in green.
- Committing your work to the provided private repository is optional, nevertheless, we strongly encourage you to regularly save your progress. Furthermore, please track your effective time spent completing each project part. This data will be very useful for future improvements of the project.

Inputs and Deliverables Project inputs you will receive consist of requirements documents (like this one) and some initial code, called a template. Templates **EventPlatformNAG** and **EventPlatformFlask** are the initially provided insecure implementations of the web application in *v*ActionGUI and Flask, respectively. **ChangeRequest** is a requirements document describing the web application changes to be implemented in the final part. For each project part, the files `project.stm`, `project.ptm` and `project.py` represent the *v*ActionGUI's security and privacy models, and the Flask implementation, respectively, that need to be submitted. Optionally, for Flask implementation, additional Python libraries must be specified in the `requirements.txt` file submitted along with any new `.py` files you create (used by `project.py`).

Project Schedule is shown in the table below. Each row represents a part of the project, while columns describe each part's start date (**Release**), the template to use and the requirements document (**Inputs**), the final code to submit (**Deliverable**), and the submission date (**Deadline**). The project must be completed individually. For Parts I and II, we will divide you into two groups (A and B). Students in group A will use the *v*ActionGUI tool in Part I and Flask in Part II, while students in group B will use the tools in the opposite order. The template you receive will determine your group. Details on the templates and deliverables are in Section 4. The deadline for each part is ~~at the end of the specified date (Zürich time).~~ also specified in the table.

	Release	Inputs	Deliverable	Deadline
Part I Group A	06.11.2024 (00:25)	This document EventPlatformNAG template	`project.stm`, `project.ptm`	20.11.2024 (00:25)
Part I Group B	06.11.2024 (00:25)	This document EventPlatformFlask template	`project.py` (other `.py` files, and `requirements.txt`)	20.11.2024 (00:25)
Part II Group A	20.11.2024 (00:35)	EventPlatformFlask template	`project.py` (other `.py` files, and `requirements.txt`)	04.12.2024 (00:35)
Part II Group B	20.11.2024 (00:35)	EventPlatformNAG template	`project.stm`,`project.ptm`	04.12.2024 (00:35)
Part III All groups	04.12.2024 (01:00)	ChangeRequest document Template modifications	`project.stm`,`project.ptm`, `project.py` (other `.py` files, and `requirements.txt`)	11.12.2024 (01:00)

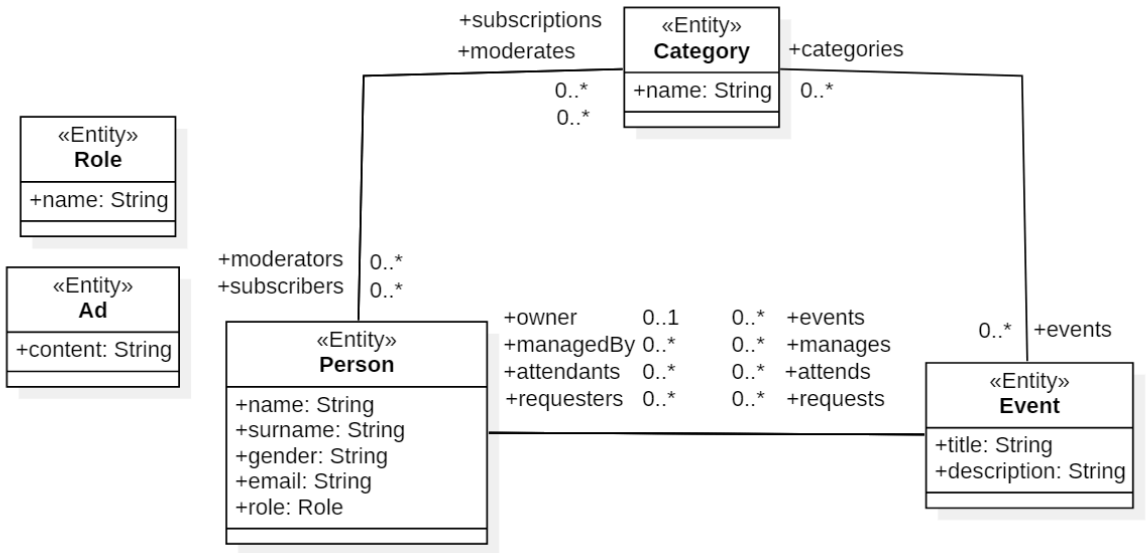
1. Event Platform Web Application

The web application to be developed is a simplified version of the typical event management platforms (e.g., Meetup or Facebook events).

In a nutshell, anyone can visit the platform and access a limited functionality. Visitors can register to be (regular) users, which provides additional functionality. The platform's content is managed by moderators, while the moderators (and the other users) are managed by the platform's administrators.

The platform's main goal is to maintain a collection of events curated by its users. Users can create events, which they then own. Events, modeled with the **Event** entity, have a `title`, and a `description`. Other users can request to attend events, attend them, and (help to) manage them. Users are modeled with the **Person** entity, which contains `name`, `surname`, `gender`, `email` (address), and `role` attributes. Events can also be grouped into multiple categories, to which users can subscribe. Categories, modeled with the **Category** entity, have a `name` and are managed by moderators. The platform serves advertisements to its users. **Ad** entity has `content` attribute only. Additionally, the platform sends mass marketing messages to its users, and provides administrators statistical information about the events.

Figure below shows a (simplified) class diagram representing the Event Platform's data model. The omitted `0..*` association end collection types can be seen in data models provided in the templates. Note that in the *v*ActionGUI data model the role attribute is implicitly added to the entity declared to represent users in the security model.



An event's associations `managedBy` and `attendants` always contain the event's `owner`, and `attendants` association also contains all the event's managers. A category's `moderators` association always includes persons with the `MODERATOR` roles. **You must maintain these invariants in your implementation.**

When entities are created, all attributes and `0..1` association ends are initialized to null (in Python, represented with None), while all `0..*` association ends are initialized to empty collections. This can make it hard to write authorization constraints in vActionGUI as one cannot reliably know which attributes have already been set to appropriate values and can, therefore, be used to express the constraint. Upon creation of each entity, you can assume that attributes and association ends will be set in the following order (in our application, we do not have functionality to create **Person**, for that reason, the order of creation for objects of these entities can be ignored):

- **Event**: `owner`, `managedBy`, `attendants`, `title`, `description`, `categories`, `requesters`
- **Category**: `name`
- **Ad**: `content`

If an attribute or association end is not listed, it may not be initialized. If it is initialized, it can happen in any order with respect to other unlisted attributes or association ends. For example, event's `owner` attribute will always be set and, in particular, before its `managedBy` association end. The `requesters` association end, on the other hand, may never be changed after the event's creation (and hence could remain empty). Hence, when writing an authorization constraint on `managedBy`, you can rely on the `owner` attribute being initialized, but not vice versa.

In this project, you will apply the concepts of model-driven security and privacy (with vActionGUI), and secure coding (with Flask) to implement security and privacy requirements described in the next sections. The provided base implementations are the starting point: they only implement functional requirements described above and do not implement the security and privacy requirements.

2. Security Requirements

We focus on authorization policies as a specific kind of security requirement. These policies consist of a set of permissions, which are described in the following subsections. Permissions are first grouped by specific entities, and then by specific roles.

Authenticated users of the event platform can have one of three roles: `REGULARUSER`, `MODERATOR`, or `ADMIN`. Anyone accessing the platform without authentication is considered to have the `VISITOR` role. In Flask, this role is not explicit, it rather corresponds to the case when `current_user.is_authenticated` returns `False`.

For the rest of this section, whenever we refer to “users”, we mean the authenticated users of the platform. Whenever we refer to “everyone”, we mean both the authenticated and non-authenticated users (i.e., both platform’s users and visitors).

General Remarks

Shared permissions

Events are managed by their owner and optionally a set of managers. For each event, managers have all the permissions that the attendants have, and the owner has all the permissions that managers have.

Association ends

In νActionGUI, for permissions related to adding, removing, and updating association ends, pay special attention to both ends of the association and implement all constraints *symmetrically*. For example, if there is a constraint for `update(owner)`, you must also define analogous constraints for `add(events)` and `remove(events)`.

Stateful Permissions

Event

Users can create events. Everyone can read an event's core information (title, description, and owner) and any event's categories. Only an event's managers can edit the event's core information with the exception of the owner association end, which cannot be changed after it has been initially set (Hint: you can determine whether the owner has been set by checking if it is null). The event's managers can edit (i.e., add and remove) event's categories.

Any user can see who is attending and who is managing an event. Only the owner can promote attendants to managers (i.e., have them be the event's managers in addition to attendants) and demote managers to attendants. Managers can remove attendants from the event, but only if the attendant is not a manager. They can also add those who have requested to join an event as attendants. Any user, except the event managers, can remove themselves from attending the event. The owner, however, cannot remove themselves from managing the event. Any user can request to join an event and cancel their requests. Only event managers can accept or deny requests. Managers can view all requests for the events they manage.

Person

Users can view any user's core information (name, surname, and role). Users can edit their own core information, except for their role. They can view their own gender and email. Each user can see the events they own, manage, attend, or have requested access to, as well as the categories they are subscribed to. The categories moderated by a user are visible to everyone.

Category

Everyone can see who moderates a category. Users can remove themselves from being a moderator of a category. Categories, their name, events, and moderators, are visible to everyone, but only category moderators can view the list of its subscribers. Additionally, users can subscribe to and unsubscribe from categories.

Ad

Everyone can see the content of any Ad but only the administrators can create, update the content, and delete an Ad.

Role Permissions

If in conflict, the permissions described here take precedence over the permissions described earlier.

`VISITOR` (**Visitors**)
Visitors can perform any actions allowed to everyone.

`REGULARUSER` (**Regular users**)
Regular users can perform any actions allowed to Event Platform's authenticated users.

`MODERATOR` (**Moderators**)
Moderators have all the permissions of regular users. Additionally, they can remove events from the category they moderate. They can also read the email of the subscribers of the category they moderate.

`ADMIN` (**Administrators**)
Administrators have the same permissions as regular users. Additionally, administrators can read any user's core information, gender, and edit any user's role. They also manage moderators and categories: they can create and delete categories, edit category names, and add or remove users with the `MODERATOR` role as moderators of categories.

3. Privacy Requirements

We focus on data protection as a specific kind of privacy requirement. Generally speaking, it requires ensuring that any user's personal data is processed according to the user's data-usage policy. Since users rarely specify their data-usage policies explicitly, data protection regulations (e.g., GDPR) mandates certain baseline data-usage policies.

In this project, we concentrate on the two key data-usage policies:

- Purpose-limitation (Art. 5 §1 (b) GDPR): Personal data shall be collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes.
- Data-subject consent (Art. 7 §1 GDPR): Where processing is based on consent, the controller shall be able to demonstrate that the data subject (i.e., the user) has consented to processing of his or her personal data.

Personal data
In the Event Platform, the following are considered personal data: the `name`, `surname`, `role`, `gender`, and `email` attributes and the `subscriptions` association end in the **Person** entity.

Purposes
For this application, we consider the following set of purposes:

- `Any`: This is the most general, top-level purpose. This purpose contains Marketing, Analytics, and Functional purposes.
- `Marketing`: This includes specific types of marketing and is divided into:
  - `Targeted Marketing`: Related to displaying personalized advertisements to users on their profile page.
  - `Mass Marketing`: Allows a moderator of a category to send a generic advertisement to all subscribers of the category.
- `Analytics`: This focuses on analyzing users' data to generate insights and statistics. Specifically, an administrator can analyze users attending an event.
- `Functional`: This represents specific functional purposes related to the Event Platform's functionality. It includes:
  - `Recommending Event`: Related to the functionality for recommending events to users on the main page of the application.
  - `Core`: Any other functionality of the platform not mentioned above is considered core functionality.

Whenever any personal data is used (actions: read, add, remove, or update) **as part of the execution** of an application's function that implements the functionality associated with a purpose above, that purpose will be considered the **actual purpose** for the data use.

> For example, suppose function `analyze` (implementing the functionality for the `Analytics` purpose) calls a function `foo` that accesses a `name` of a **Person**. The actual purpose of the data access is `Analytics`. If `foo` was called by a different function the actual purpose may be different. If a function `bar` implementing functionality associated with purpose `P1` calls a function `baz` implementing functionality associated with purpose `P2` any personal data access within `baz`'s execution will have both `P1` and `P2` as actual purposes.

In your implementation you have to keep track of the actual purpose in order to use it to check compliance to the privacy policy.

Privacy policy

A privacy policy (or notice) is a list of declared purposes (a personal data list, a purpose, and a constraint triple) shown to the user. The Event Platform's privacy policy is as follows:

- your `name` for `Marketing` Purpose.
- your `name`, `surname`, `role`, `subscriptions`, `gender`, and `email` for `Core` Purpose.
- your `subscriptions` for ~~`Recommend Events`~~ `RecommendEvents` Purpose, *if you are a Regular user.*
- your `gender` for ~~`Targeted Marketing`~~ `TargetedMarketing` Purpose.
- your `email` for ~~`Mass Marketing`~~ `MassMarketing` Purpose.
- your `gender` for `Analytics` Purpose.

Users may grant or revoke consent to each declared purpose. If the constraint is not specified in the declared purpose, the if true constraint is assumed. The state of a user's consent is stored in the database using the **Consent** entity with `classname`, and `propertyname` attributes, and the `purpose` and `user` 1-* association-ends.

To implement the Event Platform's privacy requirements one must:

1. Show the above privacy policy to the user to get their consent (see the `policy` function in Flask, or the declared purposes part of the *v*ActionGUI's privacy model)
2. Check if a personal data use is allowed for a set of actual purposes by checking if:
   - if all the actual purposes are declared in the privacy policy for that data, and
   - if the data owner consented to all of the actual purposes (or to purposes containing the actual purposes) for that data.

4. Project Parts

Parts I and II: Securing the Event Platform

In the first two parts of the project, you are tasked with implementing a secure version of the Event Platform using both the *v*ActionGUI tool and Flask. To do so, you must enforce the security and privacy requirements described in the previous sections using the provided *v*ActionGUI and Flask base implementations.

When interpreting requirements you must assume the principle of least privilege: every user must be able to access all and only the data that is necessary for them to fulfill their function. Hence, if a permission is not described or does not follow from a permission described, it must not be implemented.

Task I: Design the *v*ActionGUI security and privacy model

Objective: Write the *v*ActionGUI security and privacy model in the `project.stm` and `project.ptm` files such that the *v*ActionGUI implementation enforces the given requirements.

Template: The EventPlatformNAG contains a complete data model and endpoints of the Event Platform. It is insecure as its security model allows every action and its privacy model has no personal data and declared purposes. To ensure expected behavior, do not change the data model in the project. Note that, not all actions can be performed in the endpoints and those that can, can be implemented differently (e.g., assigning an owner to an event can be done with `event.owner = user` or equivalently with `user.events.append(event)`). Your security model implementation must be faithful to the task description, and should not rely on what the implementation in the endpoints does.

Deliverable: Submit the `project.stm` and `project.ptm` files.

Task II: Implement security and privacy requirements in Flask

Objective: Modify the implementation of the endpoints in the file `project.py` such that the Flask implementation enforces the requirements.
Template: The EventPlatformFlask is a Flask implementation of the functional requirements of the Event Platform. It is insecure as it never checks any of the permissions from the

security requirements and enforces any privacy requirements. The implementation includes a consent collection mechanism that you can rely on and your task is to implement the enforcement of the declared purposes.
You may change the `project.py` file. It contains a dictionary `PRIVACY_INPUT` that defines declared purposes and is used by `policy` as a basis to collect user consent. Make sure that you implement the dictionary according to the given privacy requirements. The `project.py` file also contains functions that implement the platform's functionality. Make sure that you maintain each function's contract: either return the (restricted) return value, or throw a `SecurityException` (for security violations), or `PrivacyException` (for privacy violations), which you may assume is handled by the Flask endpoints properly. Ensure that you identify functions that implement functionality related to different purposes, and track the actual purpose correctly. There are a few other things to note:

1. One important difference between this Flask implementation and the ones in the Flask tutorials is that we introduce a data transfer object class (DTO), for each data model class. DTOs are used to build a subset of the data model, which is to be passed to a web page template and rendered/shown to a user. By building and passing a DTO instead of a data object we ensure that the web page template can only display information the current user is authorized to see. More details are provided in the comments in the `dto.py` file.
2. Pay attention to the `Consent` class definition in `model.py`. Understanding its functionality would help you to implement the privacy checks.
3. Unlike in νActionGUI, your implementation of security and privacy requirements in Flask directly modifies the functional behavior (i.e., the function bodies in `project.py`). Therefore you do not need to account for behavior that (1) is not implemented (e.g., the Flask implementation does not provide a `cancel_request` endpoint for canceling requests), or (2) can be implemented differently (e.g., see the assigning of an owner to an event above).
4. You may use third-party libraries to implement the security and privacy requirements. In such cases, you may have to update the test setup to run the tests. You also have to submit the `requirements.txt` file specifying these Python libraries along with any additional Python files (if any).

Deliverable: Submit the `project.py` file, and optionally, the `requirements.txt` file and any new Python files (`.py`) used.

In Part I students in group A must solve Task I and students in group B must solve Task II.
In Part II students in group A must solve Task II and students in group B must solve Task I.

Submission

You will use the provided GitLab repositories to host and develop your projects. The repositories will contain the initially described templates. Make sure you commit your progress regularly. To submit your solutions you must upload the appropriate deliverables to Moodle before the corresponding deadline. **The uploaded deliverable must be your own individual work based only on the templates provided to you. Do not share your solution with anyone else, or upload it to public repositories.**

Grading

Your projects will be graded against a set of test cases designed to validate that a solution adheres to the following principles:

- **Least privilege**: every user can access all and only the data necessary for them to fulfill their function.
- **Complete mediation**: every action is checked for compliance to security and privacy policies.
- **Purpose limitation and consent:** no personal data is used for a purpose that is not explicitly declared and explicitly consented to by the data owner.

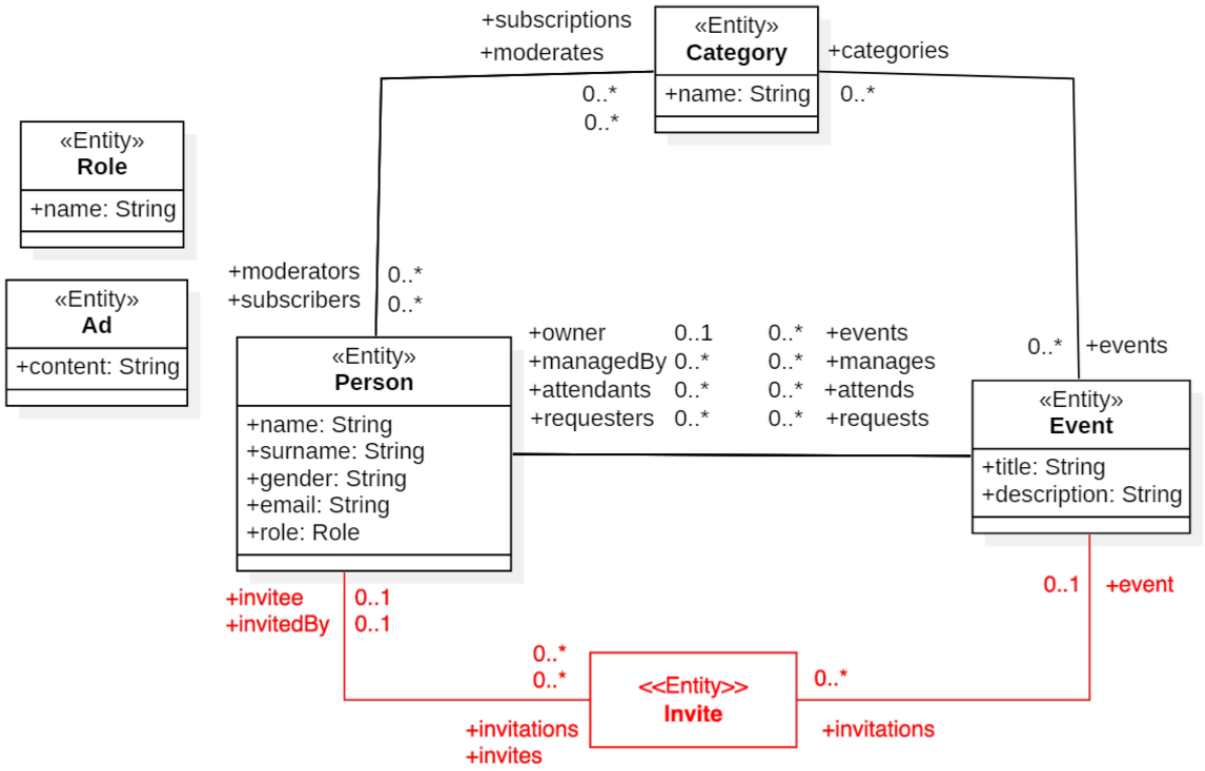
The project is worth 100 points in total. The Parts I and II are worth 80 points and Part III is worth the remaining 20 points. The 80 points are obtained by passing all the tests (both the *v*ActionGUI- and the Flask-specific ones). To obtain the 20 points in Part III, your final implementations must pass additional tests designed to check the new (security and privacy) requirements Note that there will be penalties for any new failed tests from the test suite of Part I and II. The details of the test cases will be announced later.

ChangeRequest

Hoàng Nguyễn David Basin Srđan Krstić
{hoang.nguyen,basin,srdan.krstic}@inf.ethz.ch

In the previous parts of the project you were tasked to implement a secure Event Platform web application twice, once using *v*ActionGUI and once using Flask. As a starting point, this part assumes that the functional, security and privacy requirements described in the previous parts have been implemented in both *v*ActionGUI and Flask. This part focuses on the changes to the web application to be implemented in the third part of the project.

1. Functional requirements



The Event Platform's data model must be extended to support invitations to events. An invitation is an entity that relates an event with two users: the one who created the invitation and the other who is invited to the event.

The above figure shows a class diagram representing the new extended data model. The addition of the **Invite** entity and the relevant association is marked in red. The appropriate association ends must be added to the existing entities as well as to the **Invite** entity. As an example, (i) an event can contain zero or more invitations, these invitations are in the event's `invitations` association end; (ii) an invitation is sent from one user to another, which are values of the invitation's `invitedBy` and `invitee` association ends, respectively; (iii) an invitation belongs to one event, i.e., the event is the value of the invite's `event` association end. Finally, the **Invite** entity does not have any attributes.

Upon creation of an **Invite** entity object, its association ends are set in the following order: `event`, `invitedBy`, `invitee`. When writing security permissions in *v*ActionGUI, you can assume this order. Invitations are read-only, i.e., once they are initially set, their association ends cannot be modified. It is only possible to delete the invitation.

The application is also extended to generalize the analytics functionality to insights that, in addition to administrators, provide to regular users personalized statistics about events they attend, manage, are invited to, and categories they subscribe to. Each user will have access to a page in their profile that summarizes such statistics.

We will update both of your projects to consistently change the data model and some endpoints functionality. More details are provided in the latter part of this section. To ensure that grading tests work properly, it is important that you do not change the data model or the other template files on your own.

2. Security requirements

We now focus on the extension of the authorization policy (i.e., the set of additional permissions) relevant for the invitations.

Stateful Permissions

Event

An event manager can read the list of invitations associated with the event they manage (the `invitations` association end) and read each **Invite** entity in the list.

Person

A user can read the lists of invitations they previously received or sent (the `invitations` and `invites` ~~and `invitations`~~ association ends, respectively) and read each **Invite** entity in the two lists.

Invite

To read an invitation (i.e., **Invite** entity) means that one can read all its association ends (i.e., `event`, `invitee`, and `invitedBy`).

Any authenticated user can create an invitation. Event managers can associate invitations to events they manage. They may assign themselves as the `invitedBy` for an invitation only if they manage the event associated with it. Additionally, they can invite users to events they manage by associating them as `invitee`s of their invitation. Any user can cancel invitations that they have sent. Event managers can cancel any invitation sent for the events they manage. Invitation is canceled by being deleted (in this case, its association ends are never explicitly unset).

A user who has already requested to join an event or is already in the attendance list cannot be invited to it. Similarly, if a user has been invited to an event, they cannot request to join the event.

An invited user can accept or decline the invitation (in both cases, the invitation is deleted). Any user can add themselves as attendant to an event, if they have been invited to the event. In this case, the attendant list is updated before the invitation is deleted.

Invitations are read-only, i.e., once they have been initially assigned, the values of `invitee`, `invitedBy`, and `event` association ends are not allowed to be changed.

3. Privacy requirements

We now focus on additional data protection requirements.

Personal data
In the Event Platform, the following are considered additional personal data: the `attends` association end in the **Person** entity.

Purposes
For this application, we consider the following set of purposes:

- `Any`: This is the most general, top-level purpose. This purpose contains Marketing, ~~Analytics~~ Insights, and Functional purposes.
- `Insights`: Includes the existing Analytics purpose and a new Stats purpose.
  - `Analytics`: This focuses on analyzing users' data to generate insights and statistics. Specifically, an administrator can analyze users attending an event.
  - `Stats`: This provides personalized insights to the user on their profile page.

Remaining (complex and basic) purposes stay the same. Functions managing invitations are considered to implement functionality related to the `Core` purpose.

Privacy policy
A privacy policy (or notice) contains the following additional declared purposes:

- your `attends` for `Functional` Purpose.
- your `name`, `subscriptions` and `attends` for `Stats` Purpose, *if you attended more than 2 events.*

4. Template

The update template contains changes to the data model of both *v*ActionGUI and Flask implementations consistent with what is presented in the above figure. In Flask, the update additionally extends the DTOs to accommodate the new **Invite** entity.

In both projects, we have added the options to view personal statistics (using the `personalized_stats()` endpoint) and see received invitations (in the `profile` template page). Users can also accept or decline invitations (using the `accept_invitation()` and `decline_invitation()` endpoints). In Flask, the `profile` template page traverses the additional invitations association end of the PersonDTO (rather than **Person** entity). Your implementation of the `profile` endpoint in `project.py` must be adjusted to support this.

In both *v*ActionGUI and Flask, the corresponding functionality has been added to the `manage_event` template page (In particular, we include the list of all users that can be invited, i.e., invitees, as an input to the template). Your implementation of the `manage_event` endpoint in `project.py` of Flask must be adjusted to support this. Endpoints `send_invite`, `accept_invitation`, and `decline_invitation` have also been added to the data model (in *v*ActionGUI) and their implementation to `project.py` (in both *v*ActionGUI and Flask).

The `accept_invitation` (`decline_invitation`, respectively) endpoint takes an id of an invitation as a parameter and accepts (decline) it. The `send_invite` takes an id and e as parameters corresponding to a user and event ids, and it creates an invitation on behalf of the current user for the provided user and the event.

Note: Since the database schema is updated, you need to remove the previous database schema. In the case of *v*ActionGUI, remember to regenerate the application (still using the `-re` tag).