Technology:

Using Ruby on Rails, we were able to take advantage of its model-view-controller architecture. We were able to generate a separate model for each of the main functionalities we needed, and it helped us provide methods to create, read, update, and destroy elements in each of their respective tables. The decisions of what models were made primarily based on what we needed to store - admins, participants, enrollment (who is in what course), courses, and parq (a mandatory questionnaire). We were also easily able to get started on our project, as Ruby was able to set up the main structure for each of our models for us. Another benefit of having different models was that we were able to assign different groups of people to work in different models. This allowed us to isolate and group together similar functionality, and not have to worry as much about coordinating and merging functions and files. For example, one group was assigned to work in the course model (creating, editing, and updating courses) while another group was assigned to work on managing participants (finding, and updating details). All these models were connected to each other through a static page, which served as a central hub (which we named the admin home page). This allowed for an intuitive and clean user experience, while allowing us to develop in a more organised fashion as well.

Another benefit of using Ruby on Rails was that it is, by default, setup with a database (sqlite3). Querying the database is also much simpler, as ruby has its own, more intuitive, set of commands which will auto-generate SQL statements.

Google documents were also heavily used this sprint. It was used group development of effort estimation, the product backlog, and sharing our manual testing. A google excel sheet (see attachment models.pdf) was also used to collaborate what attributes and fields would be needed in our database. This helped us agree on what should be used as keys (for when tables needed to be joined), and the format of how we were going to store different fields. This helped us save time for instances of when a group member needed to know details about another's model. They were able to simply consult the sheet, rather than inspecting the code, or having to contact the owner of the model.

Design:

There are two types of users that will interact with our system; an admin, and a member. Both users will have their own homepages consisting of all the possible models that the particular user can interact with through the controller.
So far we have only implemented the administrative components of the program, but the interface design will essentially be the same for both members and admins. When the admin initially logs in they will be directed to an admin home page.

The way the user interacts with the system is by clicking on any of the links which will cause the controller to call on the view or model accordingly. For example if the admin clicks on the "Add a member" link the controller will call on the view to display the

proper html file that corresponds to creating a member, whereas if they click on the “Export data of all participants” the controller will call on the function in the model participants called “to_csv”, in order to export all necessary information.

This type of design will be the same for the member, expect that a member will have access to different views, and models options. For example, a member cannot add another member, but they will be able to register in different classes therefore will be able to click a link that will send a request to the controller that will call on the view to display all the classes available to that member.

The diagram (see attachment UI.pdf) that our group put together before implementing anything for the admin functionalities (user interface, models etc), it specifies the different pages an admin can access, and which pages are accessed through other pages.