Sports Analysis and Visualisation System

Robert Dixon
Hamish Morrison
Nathan Owens
James Spyt

Supervisor: Susmit Sarkar

6th May 2014

Abstract

Sport science is a discipline that studies the application of scientific principles and techniques with the aim of improving sporting performance. After undertaking research into the subject as a class, the project was split into several key areas. Group E decided to undertake the job of managing the data storage backend, and create and maintain a set of APIs to provide other groups with controlled access to the data.

Additionally, we oversaw the production of a live-streaming skeletal animation system, which allows skeletal data from the user to be analysed and then visualised as a 3D model in the web browser in real-time.

This report details the aims and objectives, and the design and implementation of the project. In addition, it outlines the software development methodology used, and the intra- and inter-group collaboration undertaken to ensure the success of the project.

Declaration

We declare that the material submitted for assessment is our own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 7,460 words long, including project specification and plan. In submitting this project report to the University of St Andrews, we give permission for it to be made available for use in accordance with the regulations of the University Library. We also give permission for the report to be made available on the World Wide Web. We give permission for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. We retain the copyright in this work, and ownership of any resulting intellectual property.

Contents

1 Introduction **3**
- 1.1 Project Aims and Objectives 4
- 1.2 Initial Task Division 4
- 1.3 Architecture Evolution 7

2 Design and implementation **11**
- 2.1 Infrastructure 11
- 2.2 Database 12
- 2.3 Database API 13
  - 2.3.1 Users API 14
  - 2.3.2 Skeletal Storage API 15
  - 2.3.3 Smartcoach API 15
  - 2.3.4 Sensor API 16
  - 2.3.5 Web Data API 16
  - 2.3.6 Events and Achievements API 16
- 2.4 Skeletal Streaming 17
  - 2.4.1 Streaming Process 18
  - 2.4.2 Server-side 19

3 Software Development Methodology **21**
- 3.1 Team Structure 21
- 3.2 Tools 22
  - 3.2.1 Version Control 22
  - 3.2.2 Continuous Integration 24
  - 3.2.3 Facebook Group 24
  - 3.2.4 Trello 25
  - 3.2.5 Coding Sessions 25

4 Inter-group Collaboration **26**
4.1 Class-wide Meetings 26
4.2 API Documentation 27
4.3 Cross-group Coding Sessions 27

5 Ethical Requirements **29**

6 Testing **30**
6.1 API Testing 30
6.2 Streaming Server Testing 30
6.3 Wireframe Viewer Testing 31

7 Evaluation **32**
7.1 Evaluation against Objectives 32
7.2 Integration with Other Groups 33
7.3 Evaluation of the Project as a Whole 33

8 Conclusions **35**
8.1 Future Work 35
8.2 Acknowledgements 35

Chapter 1

Introduction

Performance in sport is becoming more and more crucial to the athletes taking part. It is important for them to understand how they are performing and what improvements can be made, in order to enhance their abilities.

To achieve repeated success, coaches and athletes must know and understand what, specifically, they have done to make them successful or unsuccessful. Coaches and athletes spend a great deal of time thinking about how to improve technique, and trying to understand the elements of a good performance.

The idea behind the Junior Honours project is to build a system to aid this process by making it easier to study a person's performance and technique: a system to provide feedback on key areas where the person is performing well, and where performance could be improved.

There are many disciplines within sport itself, which could be further broken down and studied. This project looks in particular at running, cycling, and cricket, as well as gym workouts; all of which require good fitness and technique. Existing tools already on the market include Fitbit, Nike+, and RunKeeper (among others), which track the user's physical activity, steps taken daily, and number of calories burned.

In addition to analysing one's performance, the system draws comparisons with professional athletes. This allows the average person to gauge their performance in relation to that of the very best. For example, users might go out for a run, and want to see how their times match up to professional athletes'.

The system also provides a wide range of social features, allowing users to interact with and keep up with their friends' activities. Users can login

with Facebook, and view their friends' fitness levels, activities, events, and other related information. Achievements and a leaderboard are also tied to the users' social accounts.

1.1 Project Aims and Objectives

The aim of the project is to design and develop a software system for analysing and visualising performance in competitive sports. The initial specification given by the module coordinator was intentionally non-specific, allowing for an open-ended, student-led project. This gave the students control to define the scope of the project, technology involved, and the level of inter-team communication required. The intentional lack of specificity in the project specification also encouraged us, as groups, to practice our elicitation skills by carrying out the necessary research in order to learn more about the topic.

A large part of the project was to evolve this specification, creating more concrete aims and objectives as a greater understanding of the subject was ascertained. Very early on, it became apparent that it would be important to ensure each group had an individual role within the context of the whole project, which it understood, and which would contribute to the project's overall success.

The overall aim of the module is to partake in a substantial software engineering project as part of a team, in which professional development techniques are used as preparation for industry practices. It was designed to provide close-to-real-world experience developing software in teams. However, the module is structured slightly different from previous years. This year, each team is required to make a unique contribution to one large system, which could potentially be released and sold in the market.

1.2 Initial Task Division

The change in structure of the module from previous years means that the first few weeks were spent determining what the system should do. Each group provided proposals describing the potential features that they would like the system to provide. Extensive research was also carried out to discover what similar products already exist on the market, and how to differentiate this system from those.

Representatives from each group convened for a class-wide meeting in which these ideas were put forward to the floor. This meeting was chaired by a member from group D, and minuted by our very own Robert Dixon. This meeting was used to determine each group's contribution to the overall system.

The overall aims of the project were defined as follows:

- The project will consist mainly of a single Android application and a web application.
- The web application will present information to the user about past performance and future predictions mainly with visual techniques.
- The Android application will use mobile device sensors to collect data about a user's performance, and this will be used to offer constructive feedback to improve performance and meet goals, for example improved fitness.
- Kinect sensors will be used to capture video of athlete's technique and this video will be processed to gain desired results.
- Other wearable sensors will be used to gather data and provide suggestions to improve both fitness and technique.
- The system will also be integrated with popular social networks.

The initial aims of the project were very broad. However, these were used as a jumping-off point for identifying the final requirements for the system. This allowed for a further, more detailed breakdown of the aims. In doing so, each group was able to choose two or three features to focus on as a group and contribute to the final system. These features are defined below and further illustrated in Figure 1.1. Groups are represented as letters A through to G.

Data Acquisition

- Data Acquisition from Peripheral Sensors – B
- Data Acquisition from mobile phones – D
- Kinect Wireframe acquisition – D
- Acquisition of other data as needed – D

Internal Systems

- Data Management and Infrastructure – E

Data Processing

- Ranking System – A
- Leaderboard – A
- Analysis of wireframes – F
- Smartcoach – B
- Linked with Professional Data – A
- Comparison to professionals – C

User Connection

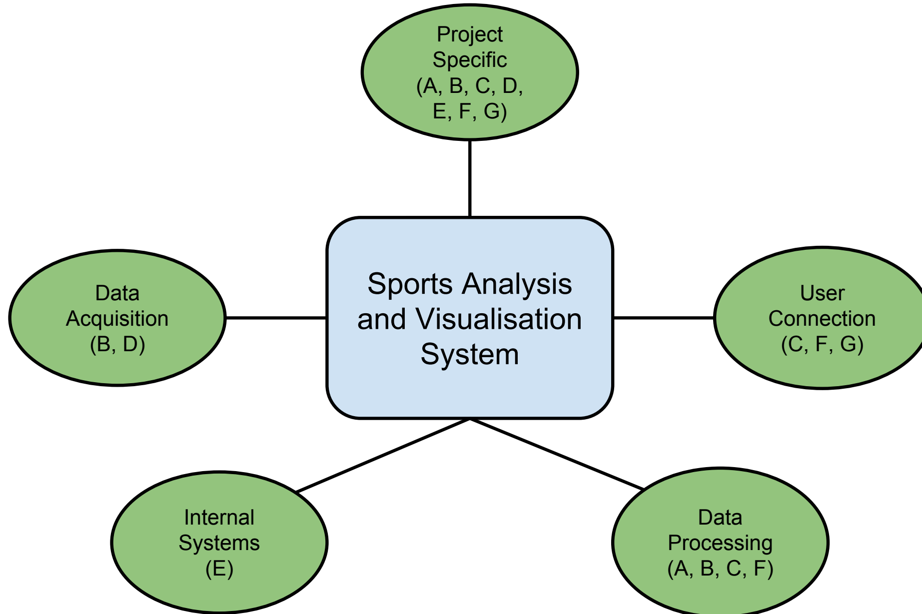
- Front-end engineering of Android Application – G
- Front-end engineering of Website – G
- General Visualisations – G
- Social Media Integration – C
- Recommending relevant sporting events/equipment using Social Media – C
- Visualisations from wireframe analysis – F

Project Specific

- Architecture oversight – E
- Ethics – All Groups

Group E assumed the tasks of database management, infrastructure design, and API development. We felt that for any large-scale system, it was important to have a central backend database for managing the system's data. It was our responsibility to ensure that access to the database was

Figure 1.1: Group Features



simple, easy to use, and easy to understand, in order to allow each group to focus on its own individual aims and objectives. A number of APIs were written to fulfil this aim. APIs were designed specifically to cater for each group's requirements. They were written to avoid other groups having to access and query the database directly, in order to improve data integrity and overall security.

The job of managing the database required interacting with each and every group. Therefore, it made sense that our group also be responsible for system architecture. It was believed that a team of architects was required, because it was vital to ensure that the system come together into a single cohesive product. The job would not only involve keeping the project on track, but also would include coordinating every presentation and demo session on account of our overall knowledge of how the system fits together.

1.3 Architecture Evolution

This section details the evolution of the overall architecture of the system through the use of a number of diagrams showing interactions between dif-

ferent parts of the system.

Figure 1.2 represents the very first model, proposed by a member of group F. It combines the initial feature proposals together into the one system.

Figure 1.2: Initial Model

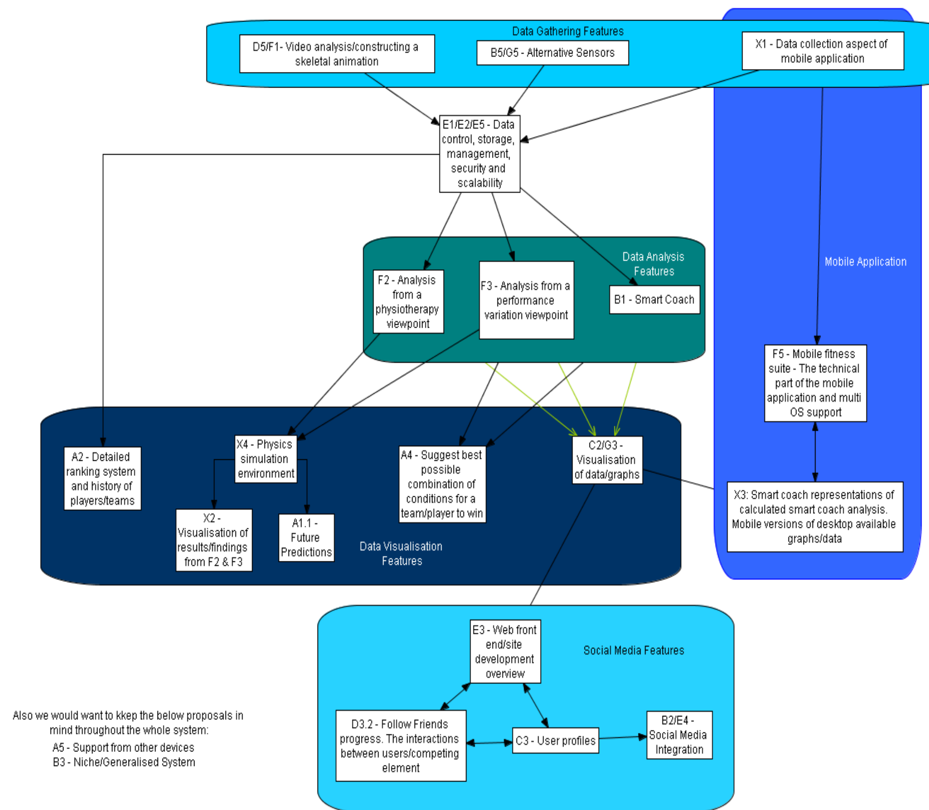


Figure 1.3 was the second model which was proposed. It was designed by our group and represents how we believed the system should be structured. Although it displayed all the system's proposed features, it intentionally did not show which groups were involved with which parts.

Figure 1.4 displays how we planned to incorporate and assign features to groups.

This plan was formalised and the result is represented in Figure 1.5.

The product came together and the final architecture is represented in Figure 1.6.

Figure 1.3: Feature Breakdown

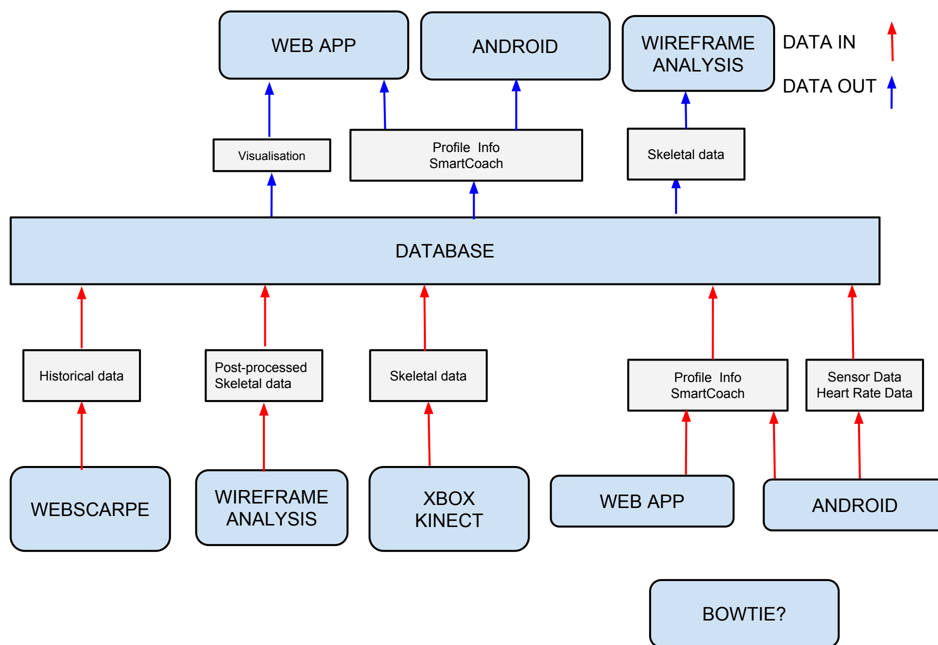


Figure 1.4: Drawing Board

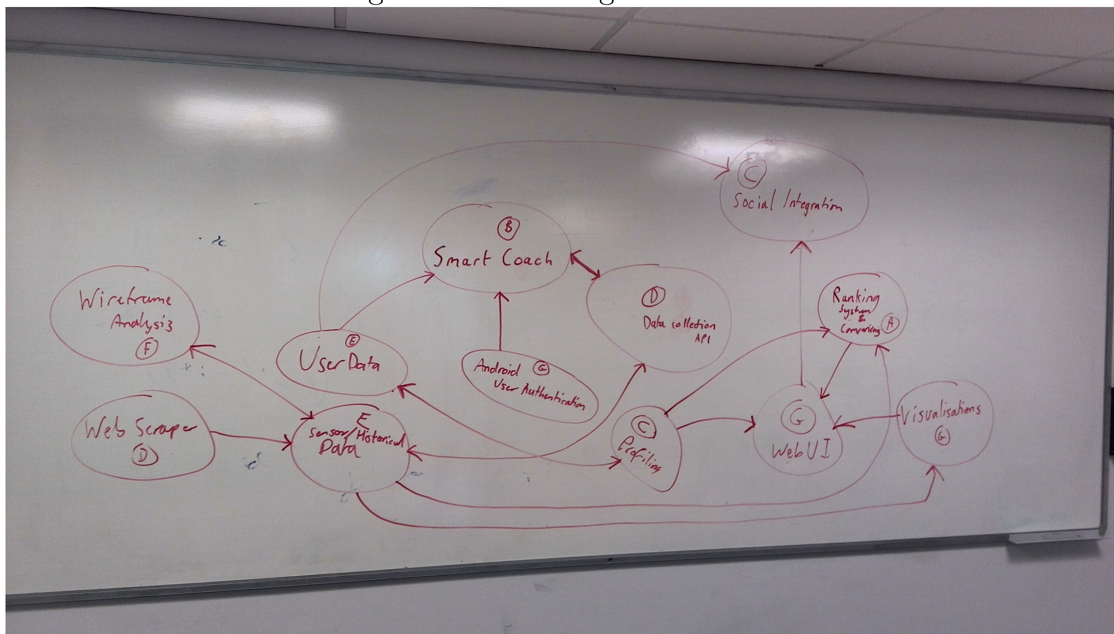


Figure 1.5: Feature Assignment

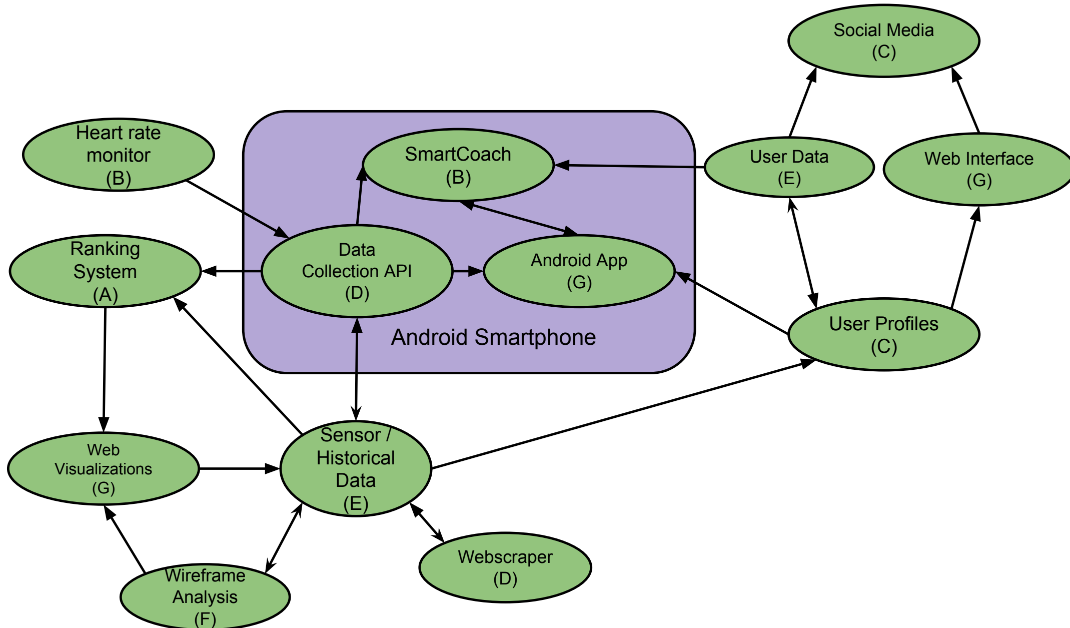
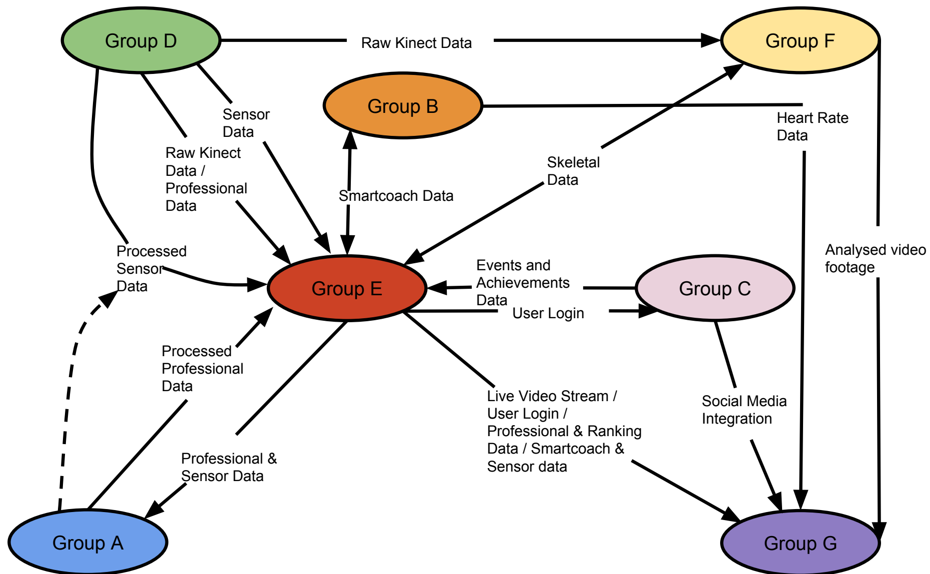


Figure 1.6: Final Model.



Chapter 2

Design and implementation

2.1 Infrastructure

From the start we intended to design a system which could easily be scaled up and would not be bottlenecked by any component. The server can be broken up into the following components:

- Nginx – a web server, the part of our system that faces the outside world. It offers superior performance to Apache because it uses an event-loop based programming model instead of forking/threads. It serves the static web data for group G on one path, and also proxies to the uWSGI instances that host the database API and the streaming server.

- uWSGI – a WSGI host for the Python web applications. It handles spinning up an instance of the Python application to handle a request when one comes in, and deals with logging and restarting servers upon failure.

- Python database API – a Python application written using the Flask framework. It provides the majority of the APIs. It talks to the database and manages the data contained therein.

- PostgreSQL (and RethinkDB previously) – a database management system. It executes queries and provides access to the data.

- Python streaming server – a WebSockets server that deals with streaming. It receives frames and pushes them to the RabbitMQ exchange.

On the other side, it retrieves frames from a message queue and pushes them to the client.

- RabbitMQ – provides a message oriented middleware for distributing the skeletal frames. Frames of a skeletal capture are routed to all consumers that have registered to receive that capture.

Although we ran all this software on a single virtual machine, different parts of the software stack can be moved out to different machines. The web-facing part (nginx, uWSGI, Python applications) can be run on multiple servers, distributing load with round-robin DNS, or using an HTTP load balancer. The database can be clustered on its own set of servers, using replication and/or sharding to distribute the load (for example, using a solution like pgpool-II). The RabbitMQ exchange can also be distributed over multiple machines using clustering or federation.

2.2 Database

The first major decision our group had to make was to choose the type of database to use. Many factors had to be taken into account including, but not limited to, the following:

- Relational vs non-relational
- What types of data were going to be stored
- How that data was going to be queried and filtered
- System scalability

Initially we chose to use a NoSQL database called RethinkDB, which was designed to store large volumes of JSON data. This decision was based on the fact that Group D would be supplying us with JSON-encoded skeletal data and sensor data. RethinkDB would have provided us with a platform for running simple analysis and queries into large volumes of data.

However it quickly became apparent that the extra querying power was not going to be used by any of the groups. Analysis of skeletal data, for example, was going to be performed a frame at a time as they were streamed, instead of over the entire capture. Additionally, the size of the data to be

stored was not excessively large, with skeletal captures weighing in at around 1MB per minute. As such, RethinkDB offered little benefit, and the lack of many features found in relational systems (relationships between tables, consistency constraints, and so on) make it unattractive overall.

At the end of January, we decided to migrate to a relational database system, PostgreSQL. This allowed us to set up more meaningful relationships between the various entities we had at the time, such as users, sessions, and skeletal captures. We were also able to greatly simplify our code with the use of an object relational mapper, SQLAlchemy. This allowed us to deal with object representations of database records, instead of hand-crafted SQL queries.

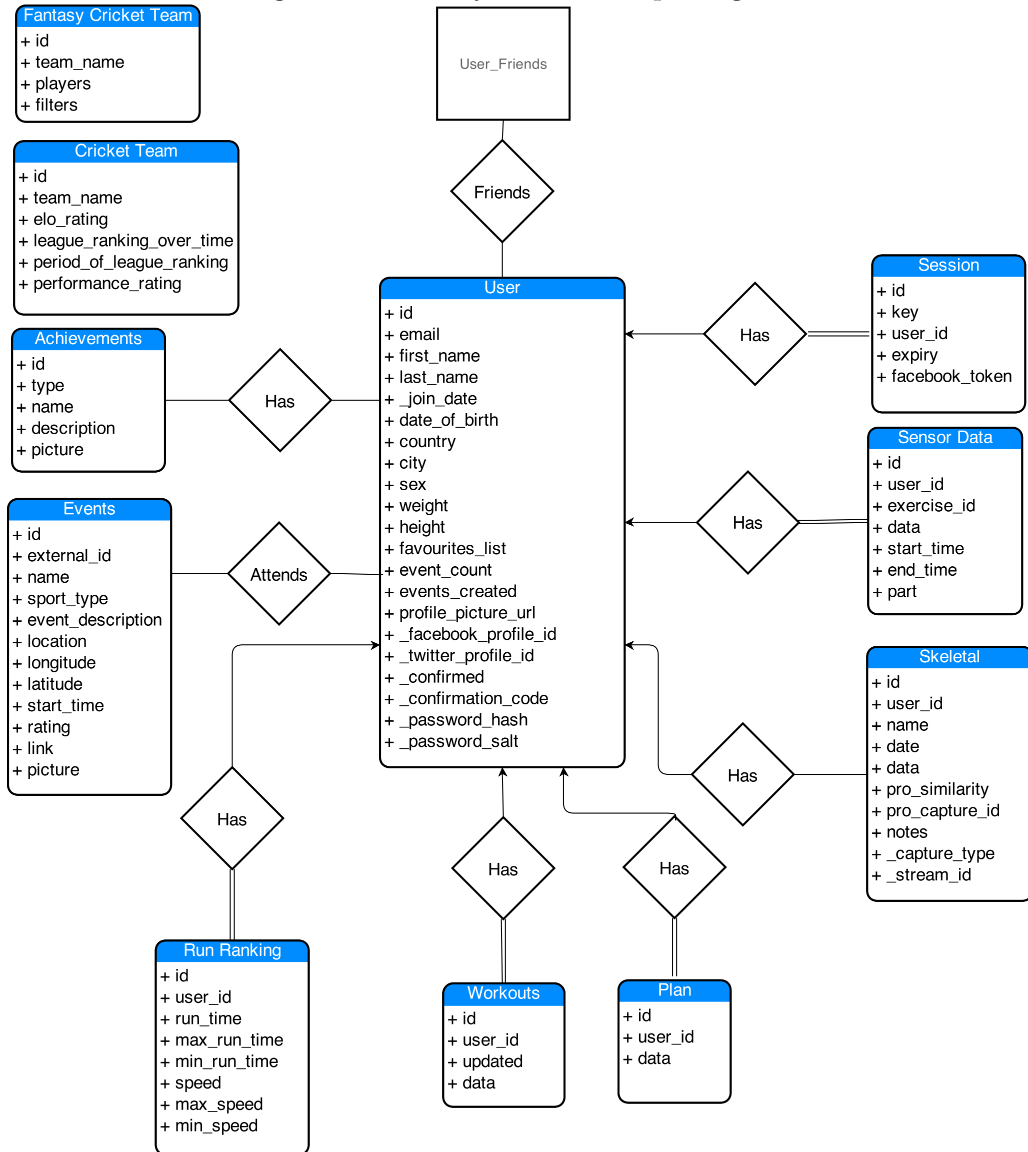
Figure 2.1 is an Entity Relationship diagram of the data stored in our database.

2.3 Database API

We decided to write the database API in Python. Each team member had prior experience with the language, and we felt that its simplicity, and the large number of third party modules available would help us write code faster. We used a web framework called Flask which provided an abstraction over Python's WSGI web application interface, allowing us to quickly set up request routing and deal with HTTP requests and responses. It also provided a facility for initialising request-local context such as the database session and the currently logged in user. This greatly simplified implementation of individual API calls, as they could assume that the database session and user context were already set up.

After writing several APIs we noticed some common patterns in them; they generally involved creating, updating, viewing, or deleting an instance of some resource, e.g. a skeletal capture or a smartcoach workout. To this end we developed some generalised methods for generating these APIs. Using reflection, we examine the SQLAlchemy model classes and validate the user parameters against these. We generate useful error messages for missing parameters or if parameters are of the wrong data type, etc.

Figure 2.1: Entity Relationship Diagram



2.3.1 Users API

The users API provides account registration, user profiles, and authentication for the rest of the system. The web front-end and the Android application use the registration method to allow users to sign up for an account. The server

handles sending a confirmation email for users to verify their accounts. The login method returns a session key which can then be used to authenticate the user when calling other APIs. The users API also provides a way to store profile data, e.g. gender, age, favourite sports, and so on. This information is used in other parts of the system, such as group C's social media integration and group A's data analysis.

At the request of group C, we integrated Facebook login into the users API. When the user logs in with Facebook, we use the email address of their Facebook account to either link the account with their existing sports system account (if they have one) or create a sports system account and link it. We provide the Facebook access token back to the social media code so they can query Facebook APIs for the user's events and likes.

We also attempted to provide a Twitter login but as Twitter does not provide the user's email address to API consumers, we were unable to link the accounts as we had done with the Facebook integration. Group C had other priorities at the time, so we simply provided the Twitter token for them to make calls to the Twitter API, and we did not pursue any solutions to the account linking problem.

2.3.2 Skeletal Storage API

The core of the skeletal streaming functionality is discussed in the next section. Within the main API, there are methods for dealing with stored skeletal captures. Professional captures are listed with a short description of what exercise and sport they depict. The web front-end uses this API so the user can pick a professional capture for comparison. The user's stored skeletal comparisons are also listed and we provide a method for naming and adding notes to a comparison. This is used by the front-end. The storage API also provides a mechanism for downloading the frames of a professional capture or comparison, which is used in the skeletal viewer.

2.3.3 Smartcoach API

The Smartcoach API provides the Android application with the ability to create and modify workouts and plans. Originally we planned to use a more relational model for the Smartcoach data, with meaningful columns. However, to expedite development while the Smartcoach data requirements were still volatile, we decided to store a lot of the data as a JSON blob. The

Smartcoach group also made some modifications to the model which further detracted from the relational structure of the database. This is clearly suboptimal, and we would have liked to model the data more clearly in the database.

2.3.4 Sensor API

We provided a sensor data storage API so sensor data could be uploaded from group D's sensor capture API which was used as part of the Android application. The sensor data is then consumed by the data analysis group, for run analysis. The sensor data API was also used by the mapping and plotting code developed by Alex Wallar, but we later integrated this code directly into the server and modified it to use the database directly. Like the Smartcoach data model, sensor data is stored in a JSON blob rather than modelled relationally. This was done for similar reasons as for the Smartcoach API, and was also never resolved.

2.3.5 Web Data API

We originally modelled the scraped cricket and running data in a relational fashion. However, the data provided by group D was not clearly structured, with the individual who wrote the scraper being completely clueless as to the meaning of what he had given us. Therefore we decided to store data for each sport as separate JSON blobs.

2.3.6 Events and Achievements API

The events and achievements API was provided for group C's social media integration project. It includes APIs for internal use, e.g. creating events and awarding achievements to users, and user-facing APIs for use in the front-end, e.g. viewing achievements and allowing users to join events and so forth. Also included is code for generating event suggestions based on user preferences and location, which was written by Dale Whitaker of group C. Since group C provided us with very well defined requirements, we were able to model it in a clear relational way.

The social media integration is problematic because the internal APIs are used from client side JavaScript code, so the API calls are exposed to anyone who knows how to look in the Chrome inspector. It would have been

better if the internal parts of the social media integration were written in Python as part of the database API (and indeed this has been done for event suggestions). This can be put down to poor communication between our groups, and lack of understanding of precisely what they were developing.

2.4 Skeletal Streaming

One of the most ambitious features of the system has been the implementation of a system to visualise animations of user's movement in real-time in the web browser. This visualisation takes place alongside that of a professional athlete for the purposes of comparison and improvement of technique. This feature uses cutting-edge technologies, such as WebGL, WebSockets, and the Microsoft Kinect SDK, which were not available just a few short years ago.

The implementation of this feature was undertaken by several groups in several different parts:

- Capture of Skeletal Data – Group D
  Group D was responsible for the production of a Microsoft Windows application, using the Microsoft Kinect SDK, to capture the skeletal data from the user and send the data to the server.

- Streaming Mechanism – Group E
  We, Group E, were responsible for the design and implementation of a system to receive wireframe data in real-time from the Windows application, and send it to the web browser through WebSockets, and organising session information between the independent components.

- Architectural Oversight – Group E
  As the feature contains so many sub-components and contributors, much oversight was required to ensure that each of the sub-components are compatible with each other and formed a cohesive system.

- Analysis of Skeletal Data – Group F
  Group F was responsible for the production of a Python module to analyse the movement of the skeleton as new frames were received, to provide informative real-time feedback to the user.

- Visualisation of Skeletal Data – Group F
  Group F was also responsible for building a visualisation of the skeletal data, using WebGL, to be displayed in the browser.

- Display of visualisations – Group G
  Group G was responsible for integrating the visualisations into the web site.

2.4.1 Streaming Process

In live-streaming the animation data, the process is as follows:

- When re-visualising previous animations, both the professional animation and the previous user animation are obtained directly from the database (through the skeletal storage API), without the need for the live analysis.

- When users access the website, they are prompted to select from a list of professional animations. This list is acquired by calling the relevant API, which queries the database for professional streams. The user selects the name of the exercise they which to practice, such as "right jab", and this is associated with an ID number for that professional animation.

- The website then launches the Windows application, passing the session key and selected professional animation ID as command-line arguments. In order to achieve this, the Windows Application registers a custom URL protocol, `skel:` in the registry, and the website navigates to `skel:<session_key>,<professional_id>` in a hidden frame.

- Once the Windows application has launched, it connects to the server using WebSockets, and sends the session key and professional skeleton ID. This then causes the server to register a new stream in the database, associated with the relevant user account (as identified by the session key), and with the relevant professional skeleton. Thereafter, it continually sends JSON-encoded frames (list of coordinates for each joint in the skeleton).

- As the server receives a frame, it is passed to the Python module group F produced for analysis. There, colours are added to the JSON object

to indicate force on joints, and a new JSON-encoded frame is outputted, and sent to a queue of frames to output, unique to that specific stream.

- Meanwhile, after having launched the Windows application, the website, has downloaded the animation from the database corresponding to the selected professional animation ID. It has then started to continually call an API to identify current skeletal streams associated with the user account. As soon as a new one has been created, the visualisations are started and a Websocket connection is started with the server. The website sends through the WebSockets the session key, and the server associates the connection with the correct queue of post-analysis frames. Then the server sends each post-analysis frame from the queue as an when they are available.

- The JavaScript visualisation stores the frames (for both the professional and user animation) in arrays, and advances the frame being visualised whenever a new frame is received.

- Finally, when the user closes the Windows Application, the code "END" is sent through the WebSockets, which closes the queue and causes its contents to be stored in the database, for access later. When a user selects a previous animation to stream, it is downloaded from the database just like the professional skeletons, rather than being streamed through WebSockets.

2.4.2 Server-side

The server side component was written in Python using a simple framework we wrote called Flocket (so named because it is a Flask-like web framework for WebSocket applications). Flocket provides some abstractions for routing the requests and for setting up per-request context such as database sessions, as well as abstractions over the WebSocket library being used (this was useful as we used a different websocket library for testing than we did on the production server).

The Python code talks to a RabbitMQ exchange also running on the server. When starting a new stream it generates a unique stream ID and sends the frames to the exchange with this ID as the routing key. When this stream is to be viewed, the server binds a queue to the exchange with the

previously generated routing key, and delivers any frames received in this queue to the WebSocket client.

The streaming server presented an interesting challenge from a scalability perspective. The classical way of hosting a Python web application is to spawn a new process or thread to handle each request. Clearly due to long running nature of the websocket connections this could result in a considerable number of threads being spawned.

We used a Python library called Gevent in an attempt to write a more scalable server. Gevent multiplexes multiple "greenlets" (essentially co-routines) on top of a single OS-level thread. Since each request spends a lot of its time waiting on I/O (talking to the websocket, talking to the database, and talking to the RabbitMQ exchange) the request's greenlet can be swapped out with another one. Gevent runs an event loop in the background and switches out a greenlet when it performs what would normally be a blocking operation (such as reading or writing to a file descriptor, or waiting on a timer). To accomplish this we had to make some modifications to the 3rd-party libraries we were using, namely psycopg2, the Postgres driver, and Puka, the RabbitMQ binding, so they would use Gevent friendly I/O routines.

Chapter 3

Software Development Methodology

One of the very few requirements for the Junior Honours project this year was the use of version control and continuous integration. These technologies make is more straightforward to develop large-scale software projects. This is achieved in two ways: first, it is easy to see what has changed, and second, you can always have a working build of the software. Throughout this project we have made extensive use of these technologies. Our group drove the decisions of which particular version control and continuous integration solutions to use in order to maximise ease of use and productivity.

3.1 Team Structure

Each member of the team was given tasks in accordance with his strengths. Along with the administrative/organisational work that is inherent to the role of software architects, the technical tasks were split into two main sections: everything internal to the server, and the external interactions of the server with other groups' components.

The skeletal streaming system is involved the interaction of several groups' work. This required a lot of architectural oversight, and so one member of the group was dedicated to overseeing this and working with the other groups to produce code that would interact properly with the other components of the system. One member of the group was responsible for handing the server side interactions with the skeletal system, as defined earlier in this report.

Two members of the group were responsible for implementing the APIs for access and manipulation of data, as and when requested by other groups.

Additionally, one person in the group was responsible for setting up repositories, continuous integration tools, and organising class meetings and presentations.

3.2 Tools

3.2.1 Version Control

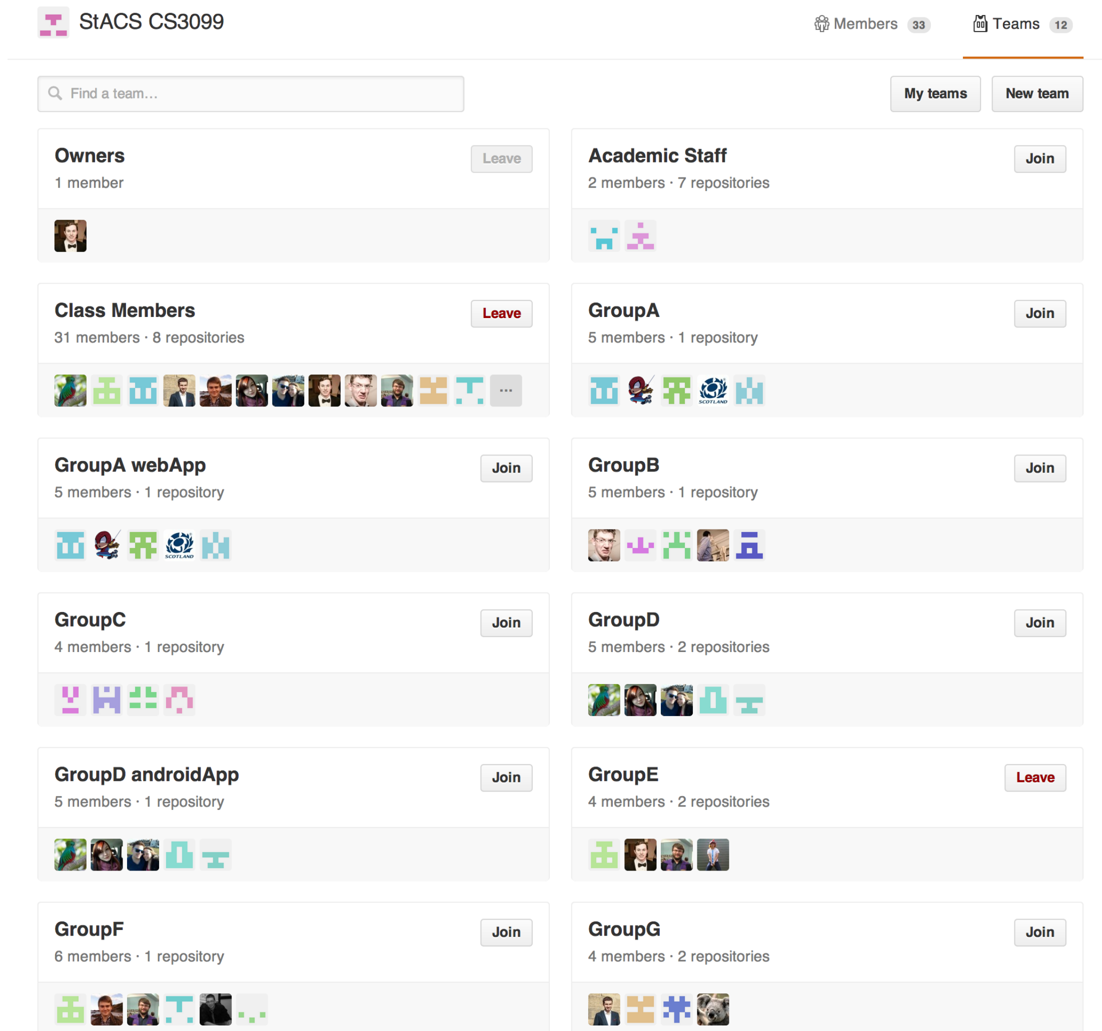
The School provides a hosted mercurial service, with a Quicksilver front-end. This is a great service for individual projects, but the web interface is very basic, and does not have any functionality for managing teams, or doing any kind of code review. For these reasons, our team decided to pursue other options. We presented to the class our proposal to use GitHub, but this was rejected as the other members of the class were worried about the reliability and availability of GitHub.

Our team went back to the drawing board and found a piece of software called RhodeCode. The website was a bit lacking, but it appeared to support almost all of the features of GitHub, but used a mercurial backend. The class approved of this idea, if we could use the School's mercurial server. We went off to deploy this solution, only to find the setup process was quite complex and it was unable to use a remote mercurial server. We again presented this to the class, and they agreed this was not an ideal solution. We proposed GitHub again, and it was then approved.

Our team created a GitHub Organisation, and added each class member to the appropriate teams. We created teams so that each member could view all other repos, but could only write to their group's repo. Once the Organisation was created, we contacted GitHub and requested ten private repos for educational use. This request was quickly fulfilled by GitHub, and we were good to go. Some members of the class were not familiar with Git, so we provided information about how to use Git to teams where needed. Our team was in charge of the GitHub organisation, and managing all of the repositories. The screenshot shown in Figure 3.1 shows all of the teams created for this project, and their members.

Using GitHub was very nice, and we did not experience any of the issues the class originally expressed with regards to availability and reliability. We

Figure 3.1: GitHub Teams.



made use of many of the features GitHub provides, such as pull requests, issue tracking, and code review. We could have made more use of these features, and, given a chance to do it again, our group would probably have suggested that all issues must be tracked on GitHub.

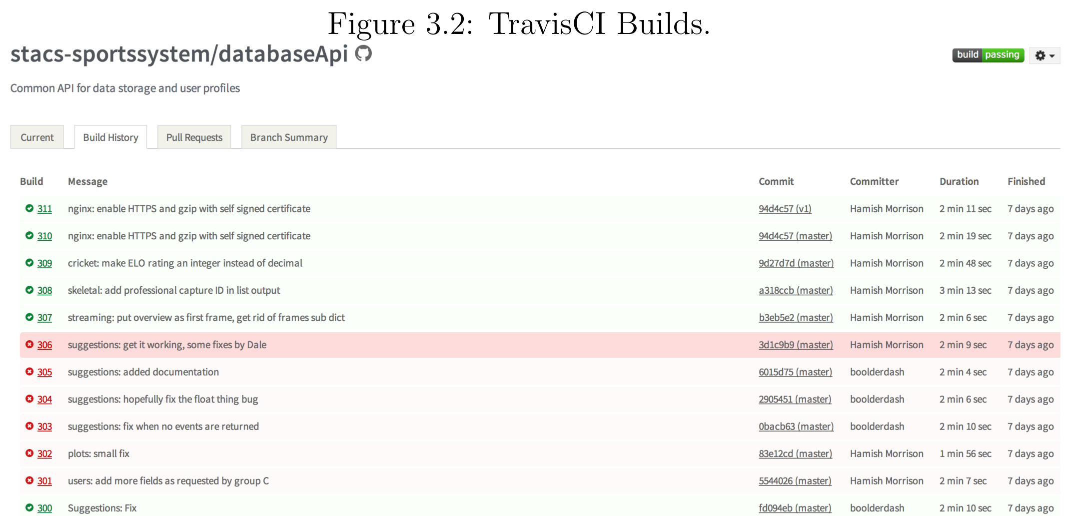
3.2.2 Continuous Integration

The School provided us with a Jenkins continuous integration server to use for this project, but the level of complication and configuration to have Jenkins work with all of the languages being used for development would have been quite high. Given our use of GitHub, our team looked into solutions that integrated tightly with GitHub. We decided to suggest a service called Travis. Travis is used heavily in the open source community, and automatically builds for every push on GitHub.

Once we presented this idea to the class, and they had approved, we contacted Travis and requested a number of private account builds. Travis, like GitHub, was very kind and granted this request promptly. Our group helped other groups develop their Travis configuration file, and managed the travis account. We added build status icons to each group's repository so they could easily see the status of their build.

In total, the class ran almost 1500 builds on Travis, and in the end each repository had a passing build. Figure 3.2 is a screenshot of our group's Travis account, showing passing and failing builds.

Figure 3.2: TravisCI Builds.



stacs-sportssystem/databaseApi

build passing

Common API for data storage and user profiles

Current | Build History | Pull Requests | Branch Summary

Build	Message	Commit	Committer	Duration	Finished
311	nginx: enable HTTPS and gzip with self signed certificate	94d4c57 (v1)	Hamish Morrison	2 min 11 sec	7 days ago
310	nginx: enable HTTPS and gzip with self signed certificate	94d4c57 (master)	Hamish Morrison	2 min 19 sec	7 days ago
309	cricket: make ELO rating an integer instead of decimal	9d27d7d (master)	Hamish Morrison	2 min 48 sec	7 days ago
308	skeletal: add professional capture ID in list output	a318ccb (master)	Hamish Morrison	3 min 13 sec	7 days ago
307	streaming: put overview as first frame, get rid of frames sub dict	b3eb5e2 (master)	Hamish Morrison	2 min 6 sec	7 days ago
306	suggestions: get it working, some fixes by Dale	3d1c9b9 (master)	Hamish Morrison	2 min 9 sec	7 days ago
305	suggestions: added documentation	6015d75 (master)	boolderdash	2 min 4 sec	7 days ago
304	suggestions: hopefully fix the float thing bug	2905451 (master)	boolderdash	2 min 6 sec	7 days ago
303	suggestions: fix when no events are returned	0bacb63 (master)	boolderdash	2 min 10 sec	7 days ago
302	plots: small fix	83e12cd (master)	Hamish Morrison	1 min 56 sec	7 days ago
301	users: add more fields as requested by group C	5544026 (master)	Hamish Morrison	2 min 7 sec	7 days ago
300	Suggestions: Fix	fd094eb (master)	boolderdash	2 min 10 sec	7 days ago

3.2.3 Facebook Group

We made heavy use of Facebook for intra-group communication. Each member of the team was a part of the project Facebook group. This allowed us to

coordinate meetings and development easily, as all group members already had and used Facebook regularly.

We also communicated with other groups using our Class of 2015 Facebook group, and through Facebook messenger.

3.2.4 Trello

We made use of Trello for some time, as a way to track which tasks needed to be done, which were in progress (and by whom), and which tasks had been completed. While Trello has a simple interface and can be quite useful, we found that having to use additional websites was more extra work than we wanted. Instead, we began to track work using GitHub issues. This reduced the total number of external services we needed to use in order to work on the project.

3.2.5 Coding Sessions

We decided fairly early on that it would be useful to get the whole group in a room to work on the code on a regular basis. Looking at our calendars, we decided to have a coding session for 3 hours every Monday. With the exception of holidays or looming coursework deadlines, we had a number of Monday coding sessions. We found this to be useful as we all had busy schedules.

Chapter 4

Inter-group Collaboration

As the database team, we were at the centre of the project and so had to be on hand at all times. Once groups had finalised their initial data requirements, it was important that we were able to satisfy these as early as possible. This was to allow the front-end visualisation group enough time to build the Android and web applications.

The main method of communication was through email which allowed requirements to be set out in full. A history of past emails gave us documents we could refer to at a later date when other groups were not available to get in contact with.

We provided documentation on our repository to let other groups know what APIs were available to access, and how to access them. This documentation was updated each time a new API was created. They specified the URL to access, indicated if authentication was required, and if any parameters had to be passed in.

For smaller issues, members of other groups were able to approach us while in labs to ask further questions or to make additions to their database requirements. This was problematic because on many occasions we had to address issues while working on practical work for other modules.

4.1 Class-wide Meetings

A number of meetings were planned at the start of the year in order to ensure a shared understanding across the groups of exactly how to proceed with the project, and to make decisions about aspects of the project that affect the

whole class. Early class meetings included decisions on which version control to use and which features to include in the final project (and which groups would be responsible for which features).

Where there was no unanimous agreement on decisions, such as which version control system to use, or which group should be responsible for the Android app, a vote was taken across the class to determine the decision.

Subsequent meetings were organised to allow for updated progress reports from each of the groups, and to give opportunities for groups to discuss their evolving requirements with the other groups. Furthermore, inter-group coding sessions were organised preceding each of the demonstrations.

4.2 API Documentation

Documentation was written to provide groups with information on how to access each of the APIs. Each time an API was completed, the corresponding documentation was written. It was important to maintain this so that groups were able to go into the documentation and find out what data they could either POST to the database or GET from the database. This limited the number of groups coming to us directly and asking for instructions on using the API.

Group G used this documentation very frequently. They were making use of all of the data stored in the database and so it was fundamental that when changes were made to this documentation, they were informed.

4.3 Cross-group Coding Sessions

We also organized a few whole-class coding sessions, in order to finalize integration. This allowed groups to have face time with our team, and ask us any questions they had about our API, or make requests for new features or APIs.

On some occasions, we had inter-group coding sessions in order to implement features that required very tight integration. We found these sessions to be very valuable for completing some of the larger features. The implementation of professional skeletal capture comparisons (as opposed to simple streaming of the user's skeleton) was one such feature, and involved members of our group, the wireframe analysis group, and the data acquisition group.

Authentication in the web frontend was also implemented as a cross-group coding effort, combining our group's knowledge of the authentication system with the web frontend group's knowledge of AngularJS and allowing us to implement the feature quickly.

Chapter 5

Ethical Requirements

As part of our role as the architecture team, and since we were the team managing all of the data for the project, we volunteered to take on completing the required ethics documentation. We worked with John Thompson, the School's ethics supervisor, to complete this documentation. We had a meeting with a member from each group to work out what our ethical considerations were, and how we could mitigate them. The majority of these ethical issues were in regards to the data storage.

The ethical concerns raised include, but are not limited to, the following:

- Data Security – what happens to the data at rest?
- Data Ownership – can the user download or delete their data?
- Data Retention – how long is the data kept for?

In order to mitigate these ethical issues, we agreed to encrypt the data when offline, but determined it was not possible to encrypt the data while in-use. We made use of TrueCrypt on the database server to create an encrypted volume for backup storage. We also made the site and API available over HTTPS to prevent any man-in-the-middle attacks, or data compromise. Other considerations pertaining to the data were also discussed, such as the ability for a user to request their data be deleted, and the length of time the data would be stored. We did not have time to implement functionality to allow the user to download their data or delete it, but these requests could be handled manually.

Chapter 6

Testing

6.1 API Testing

During development of each API, we would write a short Python script to issue some requests to the server and ensure that the results were as expected. Generally this involved adding some resource to the database, e.g. a skeletal capture or some sensor data, and checking that we were able to retrieve it again and that it was correctly associated with the dummy user. We preloaded the database with a dummy user and a dummy session so the individual API tests could focus on just testing the API in question.

We did not write any unit tests; little if any of the code lent itself to unit testing because it was heavily dependent on the database, which would have been hard to mock.

The APIs were also thoroughly tested later on in the project when other groups started to make heavy use of them and we discovered some bugs this way.

6.2 Streaming Server Testing

In developing the streaming server we created a simple HTML/JavaScript WebSockets testing page (see `static/sockets.html`). We opened this page in a few tabs on multiple machines and manually tested the basic streaming functionality, ensuring that messages got through to clients that were listening and weren't sent to clients that were listening to a different stream and so on.

Later we created a Python script to automate the streaming of a pre-recorded skeletal capture. This allowed us to test the streaming server with a more realistic volume of data, and allowed us to test the wireframe viewer without requiring use of the Kinect.

6.3 Wireframe Viewer Testing

Throughout the development process, we carried out various quality assurance assessments on the wireframe streaming system, and modified the mechanism accordingly to attain better results. For example, it was initially decided that streaming the frames into a buffer and then playing back from the buffer at the average rate the frames were received would show a much smoother and less jittery animation. Upon testing the mechanism, it became evident that this resulted in an impractical amount of delay in the video streaming, and so the mechanism was changed to advance the frame as soon as a new one was received.

Furthermore, regression testing was carried out periodically as changes were made to the design of the system. For example, when the format of the skeleton JSON data changed, these changes caused some parsing errors in the JavaScript player, which were uncovered when testing the whole system after small changes were made to one part of the system.

Chapter 7

Evaluation

7.1 Evaluation against Objectives

In early October we were required to submit a requirements specification for our team's section of the project. We specified requirements in a number of areas including: a database system, a RESTful API, and various requirements relating to these such as receiving data from other groups, ensuring security, and managing the servers.

These requirements were all fulfilled by the end of the project. We developed a RESTful API that ingests data provided by other teams into a database on a server our team manages. We use HTTPS for transport security, and have a TrueCrypt partition for keeping data backups secure. We worked with other groups to develop our APIs and database models to meet their requirements.

The streaming server has a number of unresolved issues and problematic edge-cases that we did not manage to fix. The server does not deal well with misbehaving clients; if a client connects and does not send anything, the server will wait indefinitely for it instead of timing it out (we had issues with using timeouts with WebSockets because they interacted badly with the PING/PONG messages that WebSockets must send in the background).

The streaming server also buffers up the entire skeletal capture in memory so it can save it to the database at the end. This allows a capture to potentially exhaust the server's entire memory. It would be better if the server wrote the capture out to the database a little at a time, and also rejected captures of extreme sizes in order to prevent denial of service attacks.

7.2 Integration with Other Groups

One of the goals for the changes to this module was to allow for the entire class to work together on one large-scale project. We picked a very central role in this, which required interaction with every other group. This required us to hold many meetings, and exchange many emails with members of other groups. This did present some challenges with timing and organisation, especially with the Joint Honours group who had less time to spend on this project.

Integration with group A, the Joint Honours group, was also problematic because our database API and their data analysis application were developed as completely separate pieces of software. Neither group had a clear picture of how and under what circumstances the analysis code would be called and how its output would be captured. In the final project, the data analysis code has to be run manually, and submits its results to the database using a private API.

Retrospectively, it would have been better to integrate the data analysis code directly into the database API. This way the analysis routines could be called automatically when interesting things occur (e.g. a user goes on a run, or some new cricket results are scraped) and could query the database and insert its results directly.

As noted in the section on the social media integration API, we had similar feelings towards the integration with group C, where a lot of the social media analysis code runs directly on the client-side in JavaScript. This is particularly problematic because private API calls are exposed to any user who monitors the browser's HTTP activity. Some social media functionality is integrated into the database API (event suggestions) but unfortunately this is the exception rather than the rule.

In general, we should have been more proactive in finding out how other groups were planning to implement their components and how those components would fit into the final system. If we had done this, we would have been able to strongly advise group C against writing their code to run client-side.

7.3 Evaluation of the Project as a Whole

We believe that the project undertaken could have been achieved by a smaller team of people. Doing this project with more people has required us to spend

a disproportionate amount of time ensuring co-ordination across the class, rather than focussing our efforts on the development itself.

Furthermore, we feel that the lack of a well-defined specification disincentivised work, as there were no clear objectives that had to be met by the project's completion. A more specific brief, listing requirements, would have allowed us to better measure our own progress. In addition, it would have allowed us to set more meaningful deadlines for when these requirements should be met, allowing for work to be distributed much more evenly throughout the year.

Chapter 8

Conclusions

8.1 Future Work

Given additional time we would have liked to have tried stress-testing and benchmarking our API to establish where the bottlenecks are, and based on this we would have liked to look into strategies for scaling up the system. This could have allowed us to learn about web application scalability and database clustering and replication.

Another feature that one of our group's members was keen on implementing was activity recognition for skeletal captures. This would have involved processing the skeletal data and performing some pattern recognition to determine what activity the user was performing. This was floated as a potential additional task we could take on if were we to run out of work, but in the end our original tasks kept us busy for the duration.

8.2 Acknowledgements

We would firstly like to thank Dr Susmit Sarkar, our project supervisor, for his guidance and support throughout the project. We would also like to extend our thanks to Dr John Thomson for his assistance with the required ethics documentation. Finally, we would like to thank GitHub and TravisCI for their generous contribution of free private repositories and build servers, respectively.