SAP Sailing Analytics

Architecture Overview

2012-08-03, Axel Uhl

# Introduction, Project Background and History

The SAP Sailing Analytics are a technology show-case demonstrating SAP technologies, concepts, skills and values applied to the domain of regatta sailing. They started as a small tool primarily intended to support a commentator in his job by displaying a live leaderboard for a sailing regatta with data interesting for the commentary. GPS and wind data travel from sensors to the server where the application keeps it in memory. When a request for a leaderboard is received, the data is aggregated on the fly, performing geometric computations including wind projections and involving a virtual "advantage line" orthogonal to the wind direction.

The live leaderboard started as a web application with a Java back-end responsible for the connectivity with the sensors and providing the geometry engine, and a Python process rendering the Web UI for the client's browser. The Python process issued REST requests to the Java back-end which responded with JSON documents.

The solution was first shown at Kieler Woche 2011. At the time, it was capable of displaying a single leaderboard that showed a number of tracked races in numerical form, offering columns for overall rank, race rank, rank at a mark, and values for average speed, distance traveled, gap to leader in seconds, velocity made good (VMG), estimated time of arrival at the next mark and current speed over ground. It was prototypical in many regards but regardless was considered an improvement for the commentary. The sailors liked it too because for the first time they could see numerical evidence of their choices of speed over distance.

After Kieler Woche 2011, the architecture changed. We removed the Python engine and used the Google Web Toolkit instead to render the Web UI directly in the Java process. A first new live leaderboard with this approach was shown at the IDM Travemünde 2011 and later at the MdM Hamburg 2011 events. Over time, the solution learned to manage multiple leaderboards, combining historic race analysis with live tracking. Particularly the accumulation of historic race data will require changes in the architecture in the near future to support this use case better.

A Google Map visualization, originally intended primarily for debugging purposes, matured to a useful tool used by commentators and spectators alike, combined with charts showing wind and competitor data, and of course the traditional live leaderboard. The leaderboard itself received various enhancements over time, including data about maneuvers such as tacks, jibes and penalty circles, and additional figures such as the average cross-track error which under shifty wind conditions in some boat classes may be an indicator for the risk taken by a competitor. Some of these figures turned out to be quite expensive to compute. Therefore, in a few cases we deviated from the original approach where everything was computed on the fly upon receiving a request. Instead, the more expensive calculations in live mode now happen asynchronously in the background, and client requests are fulfilled with whatever the most current result for these figures is.

The REST/JSON APIs offered by the Java back-end have been exploited by at least two additional show-case scenarios. Already in 2011, Business Objects Dashboards displayed data extracted through these interfaces in various analytical views. In 2012, the interfaces started to be used for repeated extraction of data into a HANA database on top of which Experience UI technology is now used for visualization with sophisticated analyses.

In 2012, a mobile application to support the race committees in their functions has been developed using largely the same architecture. Although the server for this app currently runs in a separate process, it uses largely the same code base, versioning repository and build process. We plan to integrate it with the SAP Sailing Analytics soon. A first loose coupling will allow users of the mobile app to send wind data entered on a mobile device into the SAP Sailing Analytics back-end where it augments the wind-based calculations. Later, we plan to integrate the mobile app even closer so that it supports race officials in laying and moving marks, changing the course layout as well as detecting and announcing disqualifications.

The remainder of this document explains the key architectural principles on which the SAP Sailing Analytics have been developed. It is to be considered a snapshot of the status quo, as documented by the time stamp in the document's header.

# Runtime Environment

## Linux, Java, OSGi/Equinox

By and large, the SAP Sailing Analytics are a web application implemented using Java technology. The application's components are OSGi bundles running in an Equinox OSGi container. Some bundles offer static web content or dynamic content in the form of servlets. Those bundles are implemented as OSGi Web Bundles which we consider a simple and elegant way to meet web standards using an OSGi-based implementation. A Web Bundle's main extension compared to a regular OSGi bundle is the presence of a web.xml descriptor in the WEB-INF top-level folder where servlets and static content can be declared and mapped to URLs.

Our current server deployment uses a 64bit Java7 Hotspot virtual machine and runs on a 64bit Linux CentOS distribution. We have a single host (sapsailing.com) which runs a number of Java VMs, some to offer the application in different development stages (dev, test, prod, ...), some to perform specific tasks such as replicating UDP wind data to the various server processes, or a process to store data received from the SwissTiming connector durably while forwarding that data to a server VM requesting it.

The various processes run in "screen" sessions to which, once connected to sapsailing.com with an ssh client, users can gain access using the "screen -rd" command. We consider replacing "screen" by "tmux" which is much more flexible and powerful and reported to be less buggy. This will allow us to launch the processes required in tmux sessions automatically after an operating system reboot.

## Database

By and large, we use a database to recover from a server restart. Once started, most data managed by the application is kept in main memory. We currently use MongoDB as our database. Different DB instances belong to different server instances. This allows us to cleanly separate development and test data from production data.

The GPS tracking data is currently usually fetched from the tracking provider in case the provider stores it persistently. While this avoids redundancies and ensures up-to-date versions of the tracking data are used, it also creates a strong dependence upon the provider's system availability and may cause performance issues when many GPS tracks need to be reloaded after a server restart.

We therefore consider using our database also for replicated versions of at least the "archived" tracks where further changes on behalf of the tracking provider are unlikely.

## Google Web Toolkit (GWT)

The web UI is built using the Google Web Toolkit (GWT). This allows us to share code between UI and back-end and gives us the power of the regular Eclipse Java tools for code understanding, debugging and agile manipulation.

The GWT application communicates with the server using GWT RPC which, in the back-end, is implemented as a so-called RemoteServiceServlet which again is exposed by means of an OSGi web bundle. This servlet accesses the core application through an OSGi service (RacingEventService) which is hooked up in the OSGi service registry.

We try to keep important styling information separate in CSS resources which can be manipulated by web designers more conveniently than the Java source code. We balance this with the benefits of the Java sources' traceability which does not exist really for CSS resources where everything is just a string.

## Tracking Connectors

To receive GPS and wind data in near real time, some network programming becomes necessary. Depending on the technology and provider used, a combination of HTTP, TCP and UDP connections is required to obtain live data. Particularly the UDP connectivity was the reason why deploying our solution to SAP NetWeaver Cloud was and still is difficult.

We try to isolate connectors to a particular technology or tracking provider so that the core application doesn't depend on a particular provider. Among other things, this leads to an architecture in which separate bundles encapsulate the connectivity components for each provider. There are still a few minor pieces of code in the UI area where this separation hasn't been completed yet and where the UI component knows about the concrete GPS tracking providers currently supported. We have plans to change this such that simply by deploying a tracking provider's connectivity bundle the back-end picks it up and makes it known all the way into the front-end.

## Wind Sensor Connector

## Result Importers

### Kieler Woche FTP Set-Up

# Basic Architectural Principles

## In-Memory Architecture and the Database

The application generally deals with two sorts of data. One is the sensor data originally created by a set of physical sensors such as GPS trackers, wind measurement devices or the sensors included in smart phones and other mobile devices. The other is master data and meta-data captured and maintained by administrators and users, such as the leaderboard configuration data, connectivity data for the tracking providers, or official scoring results imported from external sources.

For the sensor data, the time at which the data is received is never the same as the time at which the sensor data was valid. Sensors don't predict the future but measure some present value. The transmission from the sensor to the server adds a rather unpredictable latency. In the worst case, the sensor's transmission unit fails, and the data can only be imported into the server once the sensor is back on shore. With this in mind, the server's view of reality is partial and lagging, and history may be re-written at any point in time if a sensor decides to deliver its data later than most other sensors. There is no precise synchronization across the set of sensors used at an event. At best, there is a pre-configured maximum delay at which trackers make an effort to deliver their data. However, this may also fail, for example, if trackers lack network connectivity when they would actually be due to send their data. Tracking providers may also decide to re-compute some derived data which our server receives. For example, some tracking providers send data about when they think which boat passed which mark. The provider may change this at any time, sending an updated list of mark passing times. This can, e.g., happen if the course layout was changed on short notice, and the course update didn't make it into the server in time. Once the course layout change is then updated to the system, the mark passing times will be re-evaluated, and updates to the previous mark passing times will result.

These circumstances suggest an architecture which basically records the sensor facts and dynamically aggregates all derived information on the fly. This is how we started. The more complex the rules for deriving interesting figures from the sensor data grew, the more computational resources the on-the-fly aggregation required. In particular, three algorithms turned out to be quite expensive to carry out: maneuver analysis based on the recursive Douglas-Peucker algorithm; wind estimation based on the boat tracks, assuming that boats on different tacks use roughly the same beat angle to the wind; and the average cross-track error which computes a projection of each boat's position to the wind direction.

## Caching where Necessary

## Implementation Pattern for Caches

TODO talk about FutureTask, Executor, waiting for latest results vs. using a SmartFutureCache and updating in the background, using a whiteboard pattern

## Issues with Caching

TODO talk about the problem of keeping the computation and cache invalidation rules in sync

## Approaches to Locking: "synchronized" vs. ReentrantReadWriteLock

## Scale-Out through Replication

### Master/Replica Distinction

### Operational Transformation

### Operations, Services, and Events

TODO explain how some operations are issued from the outside through service interfaces, and some emerge from events such as receiving sensor data. What's accidental, what's historic, how should it be?

### Open issues

# Development Environment

## GIT, Eclipse, GWT, Maven, Target Platform

# Other Information Sources