Site Activity Tracking

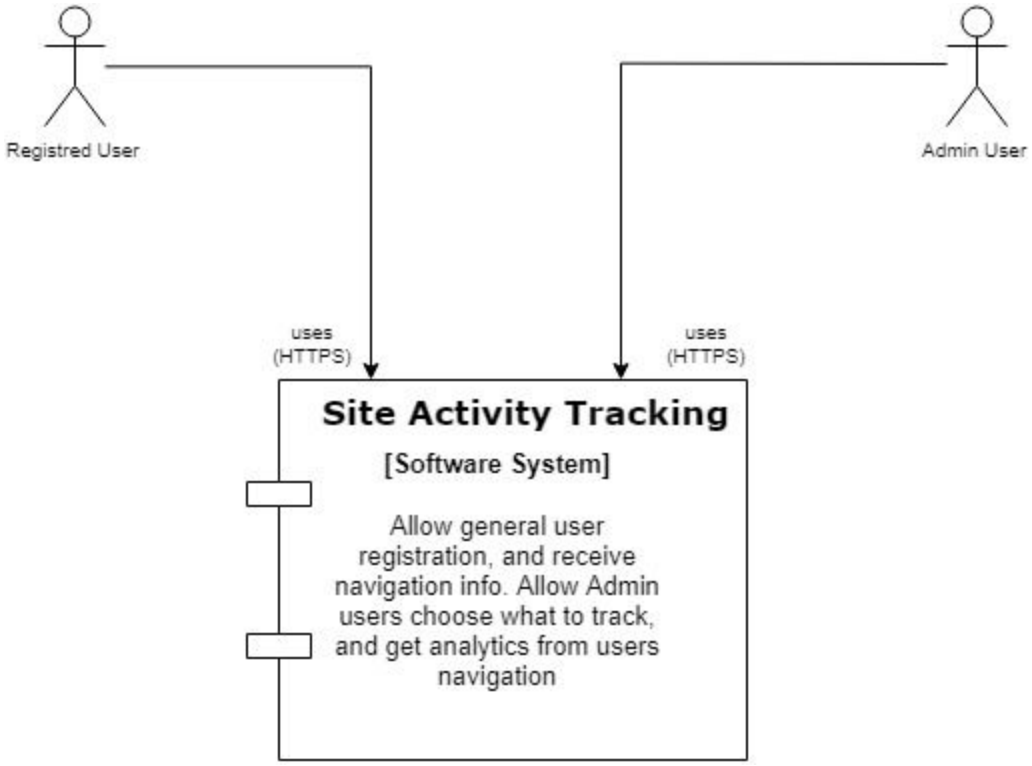
SOFTWARE ARCHITECTURE DOCUMENT
Initial version, by Carlos Filipe C. Régis, May, 2018.

Introduction

This document aims to describe the software architecture designed for the **Site Activity Tracking** system. It describes which technologies and frameworks will be used and how they were inserted in the development context. Here can be found also techniques and methodologies used on the development process, and production environment deployment infrastructure design details. We also describe some usage scenarios of the system features. The **Site Activity Tracking** is intended to monitoring of user activity on web sites. The system will be a web platform where user navigation activities arrives continuously, and administrators could extract smart insights over the collected data for the desired websites.

System Overview

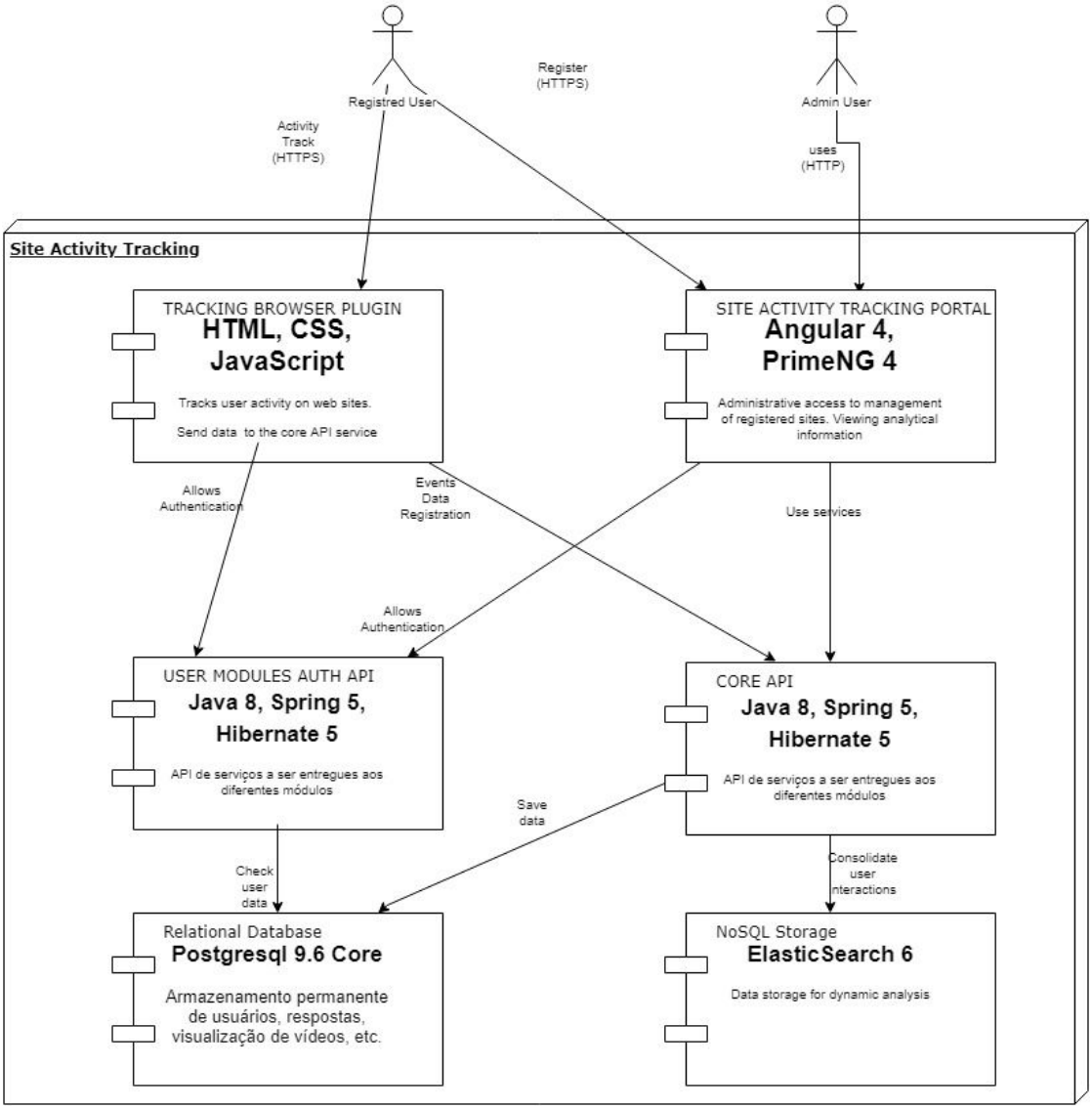
To explain in wide terms the interaction of the roles, and move drill-down to a more fine grained, this software architecture proposal document to the **Site Activity Tracking**, will use the C4 Model, to make it visible. The C4 Model, as stated by the author Simon Brown, “considers the static structures of a **software system** in terms of **containers**, **components** and **code**. And **people** use the software systems that we build.” So in the **Figure 1 - Software System Diagram**, we see the roles that interact with the **Site Activity Tracking.**



Users will self-register on the system, and then the sites chosen by the Admin users will be listened on the user browser, and notify to the server what users are accessing.

Containers view

The **Figure 2 - Containers Diagram** show in more details the containers the code artifacts or data repositories that together composite the **Site Activity Tracking** system. To implement the backend modules, the chosen language was **Java**, and the **Spring Framework** stack, because have a lot of umbrella projects, high productivity level, and wide support community. At runtime it's should be a good fit to a REST API, or even to a reactive streams flux. To deploy the **Site Activity Tracking** system infrastructure we consider to user the AWS Cloud services. To hosts code or data, the AWS solutions provide robust and scalable solutions. The same functionality could be found on others providers, but here, to this proposal, let's consider AWS services.
Here we see the proposal of 6 container components:



1. The first, the **Tracking Portal**, will be a frontend app written with Angular 4, powerful frontend build tool, and PrimeNg as components Palette to fast forms and template build. This module will communicate with the backend **Core API** to interact with the database and manage content (Admin users save sites to track and view a dashboard with usage statistics). Users will register on this module to get the tracking system running across **Browser Plugin**. This app could be served on a production environment on a Nginx server.
2. The second, the **Browser Plugin**, typically developed using HTML, CSS and Javascript, with the purpose of, once the user registered on the Tracking Portal, authenticate with the auth api, and then interact with the backend **Core API**, to send user navigation data. This module also must have a local storage to get the recorded sites that need be tracked. This local storage will synchronize periodically with a resource on the backend **Core API** to get the newest configuration available. All communication to backend Core API server will be on JSON format.
3. The third, the **Auth API**, will be a resource to get user authentication and authorization. Registered user and admin users must have credentials and limited access level to the **Core API** resources. To achieve this objectives, this module will be built using **Java** language, the **Spring Framework**, powerful stack to Java platform, in special attention to the **Spring Security** Project, with the **Json Web Token - JWT** mechanism. This mechanism encapsulate on a token a useful payload, that can be for example, bring the user information, and the authorities of that user on the system. This app could be served on a production environment on a Spring Boot application runnable jar, with a embedded tomcat server.
4. The fourth, the **Core API**, will be the main module, responsible to receive data from **Browser plugin** running instances, save this data on a NoSql Storage, the elasticsearch container. The Tracking Portal admin user actions to manage the sites will be persisted on the **Postgres** container. Also the request from real time statistics or aggregation of past data, will be modeled and processed to query the NoSql Storage and provide dashboard statistics to the Tracking Portal dashboard. To achieve this objectives, this module will be built also using **Java**, and **Spring Framework**, in special attention to the **Spring MVC** Project, to build a REST API. This app could be served on a production environment on a Spring Boot application runnable jar, with a embedded tomcat server. To save the data on the repositories, we use **Spring Data** Project, a clear interface could be found to interact with the storage services.
5. The fifth, a **Postgres** relational database management system, that will be accessible from **Auth API** and **Core API** hosting origins.
6. The sixth, a **Elasticsearch** document store instance, the NoSql Storage system, that will be accessible from **Core API** hosting origins. The collection of user navigation data will be put here, and aggregations and usage metrics will be extracted from here.

Except by the **Browser plugin** module, that is a browser running pugin, the resources of the modules: **Tracking Portal**, **Auth API**, and **Core API** will be deployed as **Docker** containers.

AWS provides the **Amazon Elastic Container Service(ECS)**, that support Docker images. The Postgres database instance could be constructed using the **AWS Relational Database Service (RDS)**, and protected inside a virtual private cloud, to be accessible from only authorized hosting origins, from our own modules. The Elasticsearch instance could be constructed using the **Amazon Elasticsearch Service**, and protected inside a virtual private cloud, to be accessible from only authorized hosting origins, similar to RDS.

To deploy the components modules, a continuous integration and delivery tool, like **Jenkins** for example, watching the source code version control repository, could trigger a system release, accessing the production infrastructure with proper credentials.
Once deployed, to productions, a monitoring tool like the **AWS CloudWatch**, that could collect and track metrics, monitor log files, set alarms, and automatically react to changes in your system resources is a powerful way to ensure high availability to the system. Application Logs of backend apps for example, could be sent to a elasticsearch document store, and a scheduled watch process could analyse for find problem patterns, and mail a support team list alerting this fact.

Development Process Techniques

To ensure the quality of the generated resources, and the validation of business goals of the systems, the following practices must be observed:

1. **BDD - Behavior Driven Development**: this is a powerful design activity, where you construct parts of functionality incrementally guided by expected behavior, aligned with the client specifications. These specifications are actually in their own domain, not in the field of software engineering. Automated test tools based on behavioral tests allow the validation of the concept developed when writing the scenarios of a feature. An example is the **Cucumber** Java library, to build automated test with the **Given-Then-When** pattern.
2. **TDD Driven Development**: build features through iterations, in which test cases are created even before the implementation of the functionality. Then, the code needed for the tests to pass must be written and the test must pass.
3. **Continuous Integration**: All automatic test steps named before could be integrated on the build construction, after every commit on the Source Code Version Repository. Continuous Integration Tools like Jenkins provide the capabilities to run test before deploy on a homologation environment. Other tools, like static code analysis SonarQube, could be integrated to Jenkins help to improve the source code product quality.
4. **Load Test**: testing that checks how systems function under a heavy number of concurrent virtual users performing transactions over a certain period of time. In other words, how systems handle heavy load volumes.The JMeter, popular one, could interact

with the built systems, in steps like obtain a user token and then send a high volume usage data, as multiple random users, for example to a large CSV file adding existing system users. On a auto scalable architecture, as **Amazon ECS**, once defined the scale parameters, the overall system should support affordably the load.

Usage Scenarios

We will describe here the initial requirements provided as BDD usage scenarios. All the feature descriptions constructed could be used in a tool like **Cucumber** to automate the test of his cenarios. After the description, a overview description of the code implementation will be given. We consider to use the **Elasticsearch** to store our massive data of user navigation, and to transform this data to another formats and generate objects to represent this insights. Let's consider the below JSON as a user access object model. This entries will be stored on a elasticsearch instance collection called **UserAccess**.

```
{
  "id": 1,
  "user": "John Thompson Jr",
  "site": "www.inlocomedia.com",
  "section": "Work with us section",
  "event": "PING",
  "lat": -8.062803,
  "lon": -34.871585,
  "date": "2018-06-25T00:00:00.000Z"
}
```

This will be the one relation on the document store that will save the user navigation data. All the other structures will be derived from this, across aggregation, average, and the suitable elasticsearch operations. Also we will store entries on another elasticsearch instance collection called **UserAccessClickBySite**, that summarize the **UserAccess** data by click, site and user by minute. A backend server task could run every minute to generate this data from the **UserAccess** collection. It's will be used on the user higher engagement identifier process feature. **UserAccessClickBySite** is in the following structure.

```
{
  "id": 1,
  "user": "John Thompson Jr",
  "site": "www.inlocomedia.com",
  "clicksAmount": 10,
  "date": "2018-06-25T00:00:00"
}
```

The **UserAccessBySite** collection will represent the aggregation of the distinct users amount on a specific site on a given minute. It's will be useful to users aggregations per minute feature.

```
{
  "id": 1,
  "userAcessAmount": 500,
  "site": "www.inlocomedia.com",
  "date": "2018-06-25T00:00:00"
}
```

Feature 1: Users aggregations per minute

BDD feature description:

```
Feature: Users aggregations per minute

  Try to get a last minute info about how many users access on site
  As administrator user on portal, go to dashboard
   So that I need is to view a panel of the sites list, containing last minute user engagement
   Scenario: List last minute user engagement by site
     Given a admin user authenticated and existing in database
     And a list of sites registered on database and a list of user access data objects with diverse users, dates and sites.
    When I enter the portal dashboard as admin
     Then the backend returns to me a list of aggregation objects containing sites, and active users.
```

Feature implementation description: We want near real time statistics of the user access data. So one good option is to build an index over the return objects of a periodic task, that runs a query on the main collection **UserAccess**. To reduce the need for an online compute process, every minute, we run an aggregation query, grouping by minute and site, counting the distinct users over **UserAccess** collection, to generate the **UserAccessBySite** object. So, when a user on the frontend portal enter the dashboard, will see the last minute computed **UserAccessBySite** list.

Feature 2: Find users access data outliers

BDD feature description:

```
Feature: Users with much more clicks than the average

  Try to get a list of users with much more clicks than the average
  As administrator user on portal, go to dashboard
   So that I need is to view a panel of the user list, containing found users with more clicks than the overall average
```

Scenario 1: List users containing clicks amount far from the overall average.

Given a admin user authenticated and existing in database

And a list of sites registered on database and a list of user access data with diverse users, dates, click events and sites.

When I enter the portal dashboard as admin

Then the backend returns to the user a list of aggregation objects containing sites, and more active users. To support the given information a users click amount overall average should be exhibited on a panel together.

Feature implementation description: one option is to build a query that answer the overall average clicks amount. This result will be used as the parameter to the second query, passing as the desired threshold, and if want outliers of more actives, the max deviation from this threshold. So we aggregate the clicks amount for each user, and filter over the max deviation from the overall average.

Feature 3: Find users highest constant engagement

BDD feature description:

Feature: Users highest constant engagement on a period on a given site

Try to get sequential period of minutes inside a week period with mor click amount

As administrator user on portal, go to dashboard

So that I need is to view a panel of the users list, containing last week minutes period with greater click engagement by user

Scenario 1: List last week user clicks greater engagement period

Given a admin user authenticated and existing in database

And a list of sites registered on database and a collection of aggregated data by click, site and user by minute.

When I enter the portal dashboard as admin

Then the backend returns to me a list of greater engagement objects containing users minutes period on last week,and the total clicks.

Feature implementation description: to find the greatest engagement period on a week, assuming that every minute have a **UserAccessClickBySite** object representing the user clicks amount on our storage, we need do the following steps: First group the sequence of minutes objects, splitting by sequence groups, using as delimiters **UserAccessClickBySite** objects where no clicks were found, `clicksAmount = 0`. After, the next action is to find the average `clicksAmount` of each group, and then find the greater `clicksAmount` average. So the group with the greater average represents the greatest engagement period to a user.

Useful Links Reference

https://c4model.com/

https://spring.io/projects

https://aws.amazon.com/ecs/

https://aws.amazon.com/cloudwatch/

https://aws.amazon.com/rds/

https://aws.amazon.com/elasticsearch-service/

https://www.docker.com/

https://cucumber.io/

https://jenkins.io/

https://www.sonarqube.org/

https://www.elastic.co/products/elasticsearch

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-avg-aggregation.html

https://www.elastic.co/blog/modeling-data-for-fast-aggregations

https://www.elastic.co/guide/en/elasticsearch/guide/current/aggregations-and-analysis.html