MAN FÅR INTE ÄNDRA NÅGOT HÄR NU!!!!1111

Markörshäng (Xtremt viktigt)

Andreas	------	------>>>	PRIVAT OMRÅDE
Linus	Dator 1	Dator 2	yo - d(°_°)b
Niclas			

Introduction

This chapter will give a short introduction and explain the structure of the thesis.

Background

Ericsson's lab and data center contains large complex Radio Access Technologies, such as GSM, WCDMA (3G), LTE (4G) and WiFi network equipment. These telecom nodes are used by projects within Ericsson to test the performance and reliability. The nodes in the lab are constantly changing their parameters and how they are connected with each other to accommodate different test setups.

There is an existing application, NNP (Node Network Planner) that is currently used to manage

the test equipment and projects. The current application does not meet up current and future requirements. The architecture of the current tool can not handle the number of nodes represented in the lab right now. The number of users has also grown at a fast pace the last years and the multi user support is lacking.

Ericsson is in the process of expanding its testing facilities by building three global ICT centers. One of these will be in Linköping. This makes the Node Network Planner an increasingly important tool for Ericsson.

Purpose

The purpose of this thesis is to investigate and create an architecture for a new network planning tool, suitable for the Ericsson test lab.

Our primary focus in this thesis is to evaluate architecture and technology choices that meets the requirements for an application of this kind. To make sure that the chosen architecture actually meets the requirements, we will also develop a prototype implementation.

Limitations

The end goal of this project would be a production ready network planning tool. However, creating such a system is a much bigger task than the scope of this thesis allows. Therefore we are forced to make some limitations.

The main focus is to create a new architecture with a well designed database and a prototype that can be improved in the future. We will not create a lot of different features and views but rather create some templates for further development. The most used and perhaps the most advanced features will be created by us in the project to show that the architecture and design works in practice.

We will explicitly not focus on:

- Making connections to other systems for information sharing
- Creating a lot of detailed views
- Define all parameters for every object in the database
- Support all third party applications connected to the current solution
- Import data from existing systems

Literature studies

A big part of this thesis is related to REST and RESTful design. Roy Fielding's thesis “Architectural Styles and the Design of Network-based Software Architectures" will be used when describing REST architecture. Roy Fielding has also written articles that will be used throughout the thesis.

We have also done some studies of ITIL (Information Technology Infrastructure Library). ITIL is used within Ericsson to define processes and workflows. Some definitions and wording that is

used to describe our prototype implementation uses ITIL terminology. These words are briefly described in the wordlist.

The literature used for implementation is mostly documentation for the libraries and databases used.

Method

This section describes the methods used when writing the thesis and during the implementation phase.

Analysis

An important part of this thesis was to gather the requirements of a new network node planner. Therefore, the first task was to gather requirements. We interviewed people from different departments within Ericsson. Most of the interviews was conducted with people from within the ITTE department (IT and Test Environment). We tried to interview different kinds of users of the current NNP tool. This includes managers, project leaders, test lab engineers and end users. The goal of the interviews was to get an overall picture of the use cases of the current system.

Design

With the ideas and requirements gathered from the interviews, we first planned the overall database and data structures. We discussed and sketched database tables and relations on paper and whiteboard to model the data. We started with a simple database design and iterated with the requirements from the analysis to make sure the most important aspects were covered.

During the design phase, we also researched and evaluated techniques and tools to use for the implementation.

Implementation

Some ideas from Scrum has been used to organize the work during the implementation phase. We have planned the tasks together. If the tasks were too big (more than half a day of work), we broke the task into subtasks. The tasks was then written down on post-it notes and put on a planning board. At the start of every day, we had a short meeting where we discussed the progress of the tasks and prioritized the work for the day.

Most of the time we have developed by ourselves, but we used pair programming for the trickier parts where more design decisions and thinking were needed. When we did not do pair programming, we tried to take on tasks in different areas of the project. This means that both of us had a very good idea of the workings of everything from database design to front-end and UI/UX. By not splitting up the responsibilities, we were constantly able to discuss problems with each other.

We have also been using some ideas from Test Driven Development (TDD), but not in a very strict way. Tests has mostly been written for API endpoints and resources, to verify that they

actually work from a user perspective. This allows us to experiment and add new features fast, but still be fairly confident that most things work, without spending a lot of time writing perfect unit tests.

Git was used for source code management. Picking Git was an easy choice since we had previous experience with it and it is also the preferred SCM system for new projects within Ericsson. Git has allowed us to easily develop on our own, while being able to painlessly synchronize the changes with each other, without risk of losing changes.

Analysis

In this section we analyse the existing network planner and gather requirements and needs for our application architecture.

Definitions

In order to understand how the existing NNP tool is used, it is important to be aware about the Ericsson terminology being used.

A “node” means an item in the test lab network. Examples of nodes are network switches, radio base stations, mobile phones or computer servers. Using ITIL terminology, a “node” is a Configuration Item (CI) in a Configuration Management system. We will use the term “node” and Configuration Item (CI) interchangeably.

The old NNP uses the term “node”. We will prefer the term “CI” instead, to follow ITIL best practice and to be consistent with the currently preferred terminology within Ericsson.

A NNP (Node Network Plan), STP (Site Testing Plan) and “Test plan” all refers to the same thing. It is a collection of nodes with a specific configuration that together makes up a complete network. NNP is also the abbreviation of the old tool (Node Network Planner). To avoid confusion we use STP when we refer to a “Site Testing Plan” or a “Node Network Plan”. We use NNP when we refer to the tool “Node Network Planner”.

ITTE (IT and Test Environment) is the department where the test lab is operated and the place where this thesis took place.

The current Node Network Planner (NNP)

In the beginning, planning a project was made with Microsoft Excel. As the size of the test lab increased, so did the amount of projects. The way of planning tests in Excel grew old and could not keep up with the expansion of the lab. The need of a better solution was demanded. That is when the first version of Node Network Planner came to form.

Node Network Planner 1.0 is a web based application constructed to replace Excel spreadsheets. The visual appearance of the application is also very similar to regular spreadsheets with rows and columns.

Each test plan and network node is stored as a spreadsheet without meaningful connections to each other. That means the database is as denormalized as it could be. It is also very hard to keep information updated and in sync between test plans, since the same information are available in multiple non related places.

Issues with the current NNP

From interviews we have extracted some key problems with old NNP:

- Inconsistent data
- Not printer friendly versions
- No different views for different needs/roles (Lars Hasselkvist, Lab engineer)
- Several other systems with associated data that are hard to synchronise (Bengt Franked, Way of working)
- Hard to find information about a specific object (Kim Myhrman, Lab engineer)
- Hard to interface from scripts/other tools (Johan Hjälm, Tester (GSM))
- Data is not up to date (Johan Hjälm, Tester (GSM))
- Hard to co-operate and keeping the data up to date (Johan Hjälm, Tester (GSM))

These are some of the issues with the current system according to the interviews we did. The root cause to many problems are that the same data is stored several times in the database. That means that if you change the value of an parameter somewhere the same object is not automatically updated in the other pages. The database is very denormalized and there is no guarantee for data consistency.

In one interview, with GSM tester Johan Hjelm, we learned that the testers had information about different equipment in the testing plan that were not correct or absent. Unfortunately they do not have any write access at all to the system and were therefore unable to correct the information.

Johan, the GSM tester also told us that it is very hard to interface the data in any other way than from the web interface. It is almost impossible to do any scripting to and from the database. There are some internal tools at the testing department which keeps track of usage between the testers. The information needs to be filled in manually at the moment.

At ITTE, there are also several systems with different kind of information about the equipment in the lab such as usage, billing information and physical placement. There is no way to get relevant information from any other system to be shown in NNP.

As there are people with different roles and therefore different interest in information there is a need for separate views depending on role. The system today offers no such ability. A lab engineer got a lot of parameters that a tester is not interested in. There are also different roles within ITTE where engineers work with different parts of a network plan.

Maintainers of lab equipment are often interested in finding information about a specific configuration item in the lab. Today you have to find the network plan containing that item to be able to read the current configuration, which is not very intuitive.

Before the ITTE engineers hand over the equipment to the testers they have a inspection meeting. In that meeting they go through the current status of the network plan and what it will include. Before the meeting had everyone printed the latest version of the plan and brought it to the meeting to be able to take notes. The problem is that there is no printer friendly version of the network plan at all. They only got a very compact version of the web page.

Good things with the current NNP

The old NNP grew out of an Excel sheet that became a web application. This was a big improvement, and there is full readonly access for STP:s and configuration item information without any required login. Making the data linkable and accessible is a very important property of the old NNP since it allows links to network test plans to be sent via email.

Something that is very useful for project managers is the ability to plan for future projects, without affecting the current STP. This makes it possible to allocate resources in the lab in the future. It also gives a way to communicate intended STP changes to both the test lab engineers and the testers.

A network of Configuration Items

The NNP database is all about modeling a network with various CIs. The old NNP does not represent relationships between different CIs in any meaningful way. The information about a particular CI is rather embedded within a STP. However, a CI might be shared among multiple STPs, therefore their parameters might be duplicated in the different STPs. This also makes it impossible to follow relations between different CIs.

We concluded that an important property of a new tool is the ability to represent relationships between CIs, regardless of their relationships to a STP. Rather than starting at the STP level and defining CIs there, we view all the CIs in the lab as a single connected network. In that network, each CI can potentially be related to any other CI. This makes it easy to follow and visualize relationships between CIs.

The CIs can then be part to a STP to support planning of projects and test setups. Since the CIs might be shared between STPs, their parameters will also be shared among the STP to avoid the duplication.

Timeline demand

The test lab is a place in constant change. The state of a CI changes over time: when it gets added to the lab, changes of parameters, changes of relations to other CI:s and finally it might be retired and discarded.

Changes to a CI might also be scheduled in the future. The old NNP has the ability to specify revision for a specific STP. While not the most convenient feature, it was very clear that some kind of versioning or tracking of past and future changes was necessary in the new tool.

Theory

This section gives a short background information of relevant theories and tools that was considered or used for the implementation.

ORM - Object Relational Mapper

An Object Relational Mapper is a tool that gives you a helping hand with the connection to the database. It maps tables to classes, rows to objects and columns to attributes. These objects can be used in your code to get easy access to the database.

An ORM takes care of more advanced things for you as well. For example it can handle relations between tables (objects) for you. That means you don't have to keep all the primary and foreign keys in your mind all the time.

Here is a simple example on how to get an users from the database:
SQL: SELECT * FROM users WHERE username='nisse';
ORM: users = User.query.filter_by(username='nisse')

RESTful design

RESTful design is an architectural structure for information exchange, originally defined by Roy Fielding's dissertation [REFERENS]. REST is basically a set of constraints on how communication over HTTP happens. It defines how information is retrieved and updated through an URL-structure. Since all communication happens over plain HTTP, it is easy to interface to a RESTful service from different languages.

For example:

GET: /cars/
Will get you a list of all cars from the service, including URLs to specific car resources.

GET: /cars/XYZ123
Now you have specified that you are only interested in the car with id 4. A POST or PUT request to this URL can update/change the resource. A DELETE request can delete this particular car.

The representation of a car can be different depending on the client. Some clients might prefer the car information as JSON, while others prefer XML. The preferred content type of a resource is specified in the HTTP request. JSON and XML are examples of different representations of the

same resource, but they are consumed in different ways. This allows for extensibility since new formats can be added in the future to accommodate new clients.
It is important to note that a REST resource must not map directly to a database table. This allows the API resources to be modelled in a more useful way than just providing access to the underlying data model.

Single-page Application / Single-page Interface

Unlike having the server generating HTML pages for the client, the HTML rendering can be done in browser itself, with JavaScript. A single page containing HTML and the JavaScript is initially sent to the browser. State changes within the application does not load a new HTML page from the server - the browser stays on the same page, but the content is changed. The browser still communicates with the server, but the data exchange is typically done with a structured format such as JSON or XML.

Web standards such as HTML, JavaScript and CSS have improved a lot in recent years, enabling new ways of writing applications this way, without installing additional browser plugins.

Design choices

In this chapter we will write about our design choices and how we thought about the architecture.

RESTful architecture

One of the main problems with the old system was that it was hard to retrieve data for consumption in different systems and scripts. An important requirement was to make writing custom scripts easy. Without having any special tools, our goal was that it should be easy for an engineer to write a simple shell script in a matter of minutes that can return useful information.

To solve this problem we needed to build an easy to use API. We looked at different ways of creating an API and came to the conclusion that we wanted to go with a RESTful architecture because mostly of simplicity and extensibility.

One alternative we looked at was SOAP. SOAP can be used over HTTP and is based heavily on XML and a specific format in which messages are constructed. This makes things unnecessarily complicated for no real gain compared to a RESTful architecture.

REST on the other hand relies more on the characteristics of HTTP itself and does not require specific formats to be used. Scripts can be written directly with curl and shell scripts without further programming tooling. REST also gives a clear path forward to improve the API. Since REST does not specify which formats to use, it is easy to extend the API with new formats and requirements in the future to adapt to new use cases.

Good REST design

One way to implement API's is to create one resource for every model present in the database. That is very useful in small projects where every model is more or less self sustained. In a larger

project like this that means that you have to perform more requests to get all the information you need.

Example: You would like to get a specific CI along with all the parameters and the parameter types. In our case that is three database models which results in three different requests to the API, if you go with the design to map one API to one model.

It is very rare that you are interested in the CI alone without any parameters. Because of that we have constructed the API's to include relevant extra information to the resource itself, such as parameters.

Another property of good RESTful design is to use absolute URLs to related resources in the response, rather than providing database internal IDs. [REFERENS TILL REST WORST PRACTICES]. This makes it much easier to interface with the API since the client must not know about any internal structure of the URLs to related objects.

Front-end

A main requirement was to provide a web interface to the system. We discussed two approaches to building the front end web interface.

The first alternative was to build a traditional dynamic web application where HTML documents are rendered on the server with a server side programming language such as PHP. This would mean duplicating a lot of functionality in the HTML documents and in the REST API.

The other alternative was to directly interact with the REST API from the browser, and build the entire application in client side JavaScript. This requires more code to move from the server side to the client, but it also makes it easier to build more dynamic applications.

We decided with alternative two to maximize the code reuse of the REST API. By creating a REST API that is also used directly by the web application, we were able to put a lot of effort into making the API easy to use. The REST API truly becomes a first class citizen when it is the one way to interact with the system.

For the task we decided to use AngularJS from Google. AngularJS is a JavaScript framework that helps you extend your HTML to handle data more dynamic. The framework also introduces a MVC structure to your web applications. It suited us well because of how easy it is to fetch data from external resources and deliver it through HTML-templates to the user. The framework keeps track of the browser history and do not reload the page but rather updates the data declared in the templates.

Client side JavaScript frameworks

One of the most important requirement for the web application is to be able to retrieve data through API's and make it visible for the user.

To do that without having to implement a whole front-end framework, we could have used jQuery. That is a JavaScript library that makes HTML manipulation easy and gives you a simple way to get new data through Ajax calls.

Choosing jQuery would give us a quick start in building the web interface. It is really easy to get everything going in the beginning but as the project grows the overhead for every new feature do as well. Because jQuery is a library that provides easy-to-use methods for a variety of usage but do not offer a complete solution for a front-end platform you have to glue everything together by yourself.

We figured that just using jQuery would cause a lot of extra work for us in the end. The way to go for us would be to use a complete framework. There were two frameworks that were on our list. One was Backbone JS and the other was AngularJS.

Backbone uses jQuery to do AJAX calls to fetch new data. You still have to manipulate the DOM if you want to alter the content in the templates after the initial rendering.

AngularJS is like Backbone a framework that gives you a lot of help for free. It also adds some structure to your application by introducing the Model View Controller concept. With Backbone you know where to place your code depending on the purpose of it. That is the main benefit you got over using jQuery alone. Angular gives you a lot of help with the code structure even though it is more up to the developer how to structure it.

Angular does also provide a smarter way to present data to the user with smart functions built in the views. The views and controllers share an information layer, called scope, where information can be stored. If the data in scope changes it will trigger the views to update the page for the user, without reloading the page.

We decided to go for AngularJS as it provides a lot more functionality straight out of the box that we needed. Instead of having jQuery replacing a lot of HTML after every API request we could just assign the data to the scope layer and Angular would take care of the rest. No manual DOM manipulation needed.

Back-end

As for back-end we aimed for something that could be of great benefit for building RESTful resources. We wanted something that could handle routes for us and map the routes to our RESTful controllers.

One of the requirements on the back-end was that it should be able to provide a powerful Object Relational Mapper (ORM). To be able to retrieve and update information in the database is one of the key functions in the project. The opportunity to be able to automatically manage database resources through objects will save both a lot of time and headache. By using an ORM there is

no need to write a lot of SQL queries and figure out which tables to join to get the correct information all the time.

These are the main criteria on the back-end platform. The one that suited us and the project best was Flask. That is a micro-framework built in Python. Being a micro-framework means that just the basics of a framework is included from the start. You can then add components as they are needed in the project. By using a micro-framework we removed all the unnecessary functionality and let us focus on the things that would benefit the project. One of the most important things is that there are no pre-defined methods on how to perform certain tasks. You can choose whatever method that works the best to fulfill the requirements on the product. Flask is easy to integrate with the SQLAlchemy ORM.

We also considered using the Django web framework. Django has more built-in features from the start, such as an ORM, form validation and an administrative interface. However, we felt that we would be better served by a smaller framework since we were building pure REST resources.

Database design

Choice of database

We choose PostgreSQL as the database for the project. The alternative to PostgreSQL was MySQL, which is very similar in the available features. PostgreSQL has more features than MySQL and one feature in particular stood out that we thought would be useful: native support for JSON.

PostgreSQL can store JSON data in a table column, where the data within the column can have constraints applied and be indexed. We planned to use this feature for storing parameters (key-value) for CI's.

Choice of ORM

To model the database tables, we have used an ORM called SQLAlchemy. This means that we can define tables as classes directly in Python code. This serves two purposes: It sets up the mapping between the Python classes and objects and allows us to generate the SQL schema directly from the Python code.

Apart from mapping Python objects to database tables, SQLAlchemy also generates the SQL queries. This saves a lot of typing and complexity when dealing with the database. It is possible to tune the query and specify how joins are performed in order to generate optimized queries.

Changes to the database schema is also managed by SQLAlchemy with an extension called Alembic. Alembic can compare the Python table definitions with the schema in the database and generate migration scripts that upgrade the database. This makes database changes such as new tables, new columns or changes to existing columns very straightforward.

Timeline

One of the requirements was to have some sort of revision control to keep track of changes over time. To address this requirement we assign a lifetime (start and end timestamp) to all parameters and relationships. They all get a start time from when they are active and an end time when they are no longer in use. This will provide a timeline of the entire network. It is therefore possible to get a view of the entire lab at any given point in time. It also makes it possible to plan future changes to any parameter without affecting the current state.

In the database, each parameter and relationship has a start and end “datetime” column which specifies the the lifetime of the objects.

When querying for parameters and relationships for a particular CI, we filter on the start and end column to retrieve the correct values for that particular time.

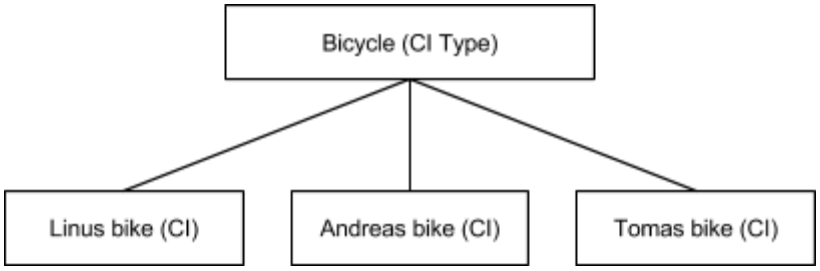
Originally, we planned to use a JSON field to store the CI parameters in a easier way, but having the start/end times along with all values caused this to be impractical, so the parameters instead have their own table in the database.

Dynamic CI types

Another important requirement was to be able to define new types of configuration items dynamically in the database, without changing the application code. When new equipment are added in the test lab, the database schema should not need to be changed or updated.

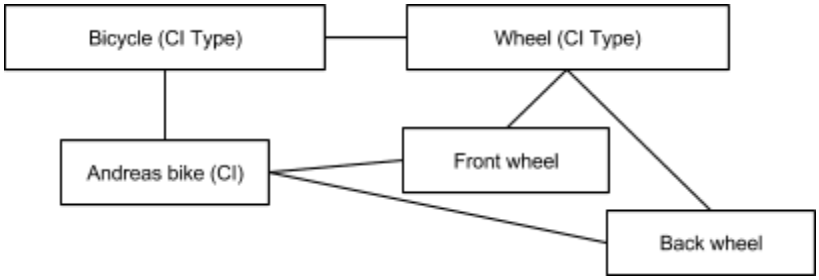
To achieve this, we modelled CI types in the database as their own objects. Every CI is then related to its own type.

This example shows how a CI type could be represented and connected to several instances:



As shown in this example, the CIs are not necessarily telecom nodes or network equipment - the CIs could be any item. Each CI type can also define any number of parameters that can be added to the CIs. A bicycle could have a manufacturer, frame size, weight and color for instance. These types of parameters would be defined on the CI type. Each CI would then be able to

specify these parameters with values. A CI can only be related to another CI where their respective CI types are related to each other.



This picture shows a relationship between the bicycle type and the wheel type. It shows how the bike and the wheels are connected. The relationship between “Andreas bike” and the wheels is possible when the bicycle and wheel types are related.

Just like parameter types can be defined on the CI type, parameter types can be defined on the relationship between CI types. Using the above example, parameters on the relationship between a bicycle and a wheel can describe parameters that are not directly related to a bike nor a wheel. Such parameters could be the type of bolt or the way the wheel is mounted.

Being able to configure CI types, parameters and relationships of objects dynamically in the database gives a lot of power to model complex networks and yet unknown equipment.

Visual design

To be able to demonstrate how to use the data from the database we needed to create a web interface. The current website has an old design with no modern style guidelines. We wanted to create a simple and clean user interface with some of Ericssons own colors to blend in with their other tools.

Multi user support and collaboration

To be able for more than one person at the time to work with a test plan we had to implement some features to make that possible.

First of all we would had to remove the need of a global save button. Instead of saving the whole test plan for every new change we only update the database with the new value of the parameter that has been changed. That minimizes the risk of overwriting each other with old changes.

To make the users aware of new updates from other users the data is automatically refreshed in the background. That means that if no changes has been made to the data, the page will stay in the same way. If the data has been altered by someone else, the browser will update the page with the correct values without reloading the page.

With these two features, the risk of overwrite each other will be drastically minimized. The only way you will be able to do that is if you are updating the same parameter for a specific CI exactly within the same second.

Views depending on role

As there are many people with several different roles working with this tool it would be convenient if the views could be customized to suit the individual needs of each user.

There are only a few roles who are interested in every item in a test plan which means that there are a lot of people working with different independent parts of the system. These people are often technicians, according to the interviews we did. It would therefore be great if they got the chance to minimize or hide parts of the test plan that are irrelevant for their day to day operation. That would highlight and give them a better overview of the parts they are responsible for and reduce waste of time searching for them.

Result

This chapter contains information about the final implementation.

Overview

Since the main goal of the thesis was to come up with an architecture that could handle large amount of data and relationships between the objects, that is what we have focused on the most. On top of that there is a RESTful API which could be used for the communication with the database.

To be able to demonstrate the architecture and database structure we created a web client that used the REST API for data exchange.

In the end of the thesis we did performance enhancements and load tests to make sure that the system would still perform in a large scale environment.

Web views

The front-end consist of just a few views. A list view for configuration items and test plans as well as detailed views for both of them.

Configuration items

One of the two main views is a page where you can look at a configuration item in detail. In the view you are able to view all the parameters for a CI at a specific time. You can change the time to see how the values of the parameters have changed over time. Values can be viewed both back in time as well as how they are set to be in the future.

Each relation to a CI is also displayed along with the parameters of the relation. It enables you to easily click around between the detailed view for configuration items that are related to each other.

PRINTSCREENOS

Test plan

The other view is the test plan view. It is not so much different from the configuration item view. In short, it looks in the same way but displays all the configuration items in the test plan. In addition to the CI view there is a fixed sidebar which follows the scrolling of the page. In the sidebar you will find a timepicker and a list of all the items in the test plan grouped by type. Each item is linked within the page and will quickly take you to the CI of interest. That removes unnecessary time searching for a CI and makes jumping between two entries much faster.

The timepicker determines for which time the values of all the configuration items will be displayed. It works in the same way as for the view for a single CI.

PRINTSCREENOS

It is possible to hide one or more CI blocks which will minimize them and only displays the name of the CI. Unfortunately it is not possible to save the settings for which type of items that should be minimized.

API resources

The system ended up with a lot of RESTful API resources. Every resource returns a JSON response with the content.

Filter results with query strings

The API:s accept inputs through query strings. With query strings you are able to apply filters to your API responses. Every resource accepts different query strings depending on the functionality of the filters.

Most API resources can be filtered with a timestamp. The timestamps make the API to return the data in the way it was for that specific time. The timestamp can be now, sometime in the past or in the future. If the timestamp query string is not provided in the API call the current time will be used.

API example

Here is an example to give you a better understanding how to use the API. The example will be based on the CI resource which is the main resource in the project. From this resource you will receive jsons with CI objects.

The API looks like this: **/api/ci/<name>** where <name> is optional. If name is included in the URL you will only receive that specific object.

As mentioned in the chapter “Filter results with query strings” you can pass query strings to the API. Query strings available for the CI resource is: timestamp, ci_type, test_plan, related_ci, related_test_plan.

Here is a quick explanation of what the different query strings does:

ci_type: takes a ci_type url input and returns all a list of CI’s of the type provided
test_plan: takes a test_plan url as input and returns all CI objects that is related to a certain test plan
related_ci: takes a CI url as input and returns all CI objects that is related to the CI included in the query string
related_test_plan: takes a test plan url as input and returns all the CIs that is not related to the test plan but is related to one or more CIs in the test plan

The API uses URLs to other resources in the response. To use filter from query strings you will often pass in a URL. For example /api/ci?ci_type=http://server-adress/api/ci-type/ci_type_name. This will make the API resource filter your results based on the query string and return a list of objects of the type you declare in the query string. The other query strings works in the same fashion.

It is also possible to chain multiple query strings. So if you for example want all CIs of a specific type from a test plan you can use both the ci_type and test_plan query strings. You can combine all the query strings to achieve the result you want.

When requesting data from this API you always get a list of CI objects in return. The API resource does some things in the background for you automatically. Every CI object got their respective parameters included in the response by default. So you do not have to do separate request for pulling parameters for every object return by the resource.

Here is an example how a JSON response from the CI resource looks like:
<JSON example of ci_list>

API reference

We ended up with five API resources in total where the CI resource from the example is the most central.

Here is a short API reference of our implemented resources:

CI

Description: Handle CI requests. Returns a list of CI objects with parameters included.
URL: /api/ci/<name>

Query strings: timestamp, ci_type, test_plan, related_test_plan, related_ci

CI Type
Description: Handles CI Types. Returns one or a list of CI Type objects
Url: /api/ci-types/<name>
Query strings: -

CI Relation
Description: Handles relation data between CIs. Returns one or a list of relation objects.
URL: /api/ci-relations/<id>
Query strings: timestamp, ci, test_plan

CI Type Relation
Description: Handles the relation between CI Types. Returns a list of all CI Type relations.
URL: /api/ci-type-relations/
Query strings: -

Test Plan
Description: Handles the test plans. Returns one or a list of test plans.
URL: /api/test-plans/<name>
Query strings: timestamp

Angular API services
To be able to use the API resources in a convenient way we implemented an extra layer in Angular which would handle all the communication with the APIs. The layer consist of several services which could be injected and used in the controllers where needed.

The services were named based on the functions they provided. For example is the service communicating with the CI API is called ciService. To keep the controllers clean and simple most of the logic has been moved to the functions in the services.

The services can do multiple API calls to different resources if necessary. The ciService got the functions getList() and getListWithType() for instance. The getList functions returns a list of CI objects from the CI API while the getListWithType returns a list of CI objects including their type. The service has made two API calls to get the CI objects and the CI Type objects and then merged them together before returning the function.

With this service layer the controllers can stay the same if the source of data is changing. The data is also accessible from every controller without having to write all the calls to the API over

and over again. It is also much easier to introduce cache functions for the data if there is only one place the API calls are made from.

Performance

Database query performance

One concern was that the system would not perform well with many objects in the database. To get some information on how our system would perform with a big database, we loaded one million CIs, 10 million CI parameters and relations to the database. We then investigated the performance on the queries used in the API endpoints [REFERENS TILL VÅRT EGET SCRIPT?]. While 10 million objects in each table is not an extreme amount of data, a database query that is naive and not properly optimized will typically execute in several seconds. The most common case we found is when issuing a query that is supposed to filter out certain rows, but that for some reason are unable to use an index to limit the results. This forces Postgres to fallback to a “full table scan”, which takes O(n) time since all rows needs to be examined.

By carefully examining the query plans and strategies used by the Postgres, adding indices and changing queries, we managed to get all queries to be fast and efficient. Some queries where easy to fix: Just make sure that all columns used in WHERE clauses were indexed.

Some queries was a bit harder to fix than simply adding an index. One example is the query that is used to fetch all relations for a particular CI. First, we used a query like

SELECT * FROM ci_relation WHERE a_name=’CI NAME’ OR b_name = ‘CI NAME’

This query turned out to be very slow since Postgre falls back to using a sequential table scan instead of using the indexes on a_name and b_name as we expected. Postgres query optimizer cannot make use of the indexes along with the OR condition. To get around that limitation, we changed the query to use the UNION statement:

SELECT * FROM ci_relation WHERE a_name=’CI NAME’
UNION
SELECT * FROM ci_relation WHERE b_name = ‘CI NAME’

This achieved the same result and allowed us to properly utilise the index on a_name and b_name.

Making a single query fast is good, but it also important to make few queries. When using SQLAlchemy in a naive way, it is easy to accidentally introduce code that executes a lot of queries to implicitly fetch related objects. By default, SQLAlchemy [http://docs.sqlalchemy.org/en/rel_0_7/orm/loading.html#using-loader-strategies-lazy-loading-eager-loading] uses lazy loading to retrieve related objects. This means that if a query yields 100

objects where we also want to fetch a related object, we will end up with 1 query to fetch the list, and 100 queries to fetch each related object.

By instead using the “joinedload” strategy, SQLAlchemy will issue a SQL JOIN statement that will fetch all related objects with the same query. SQLAlchemy has multiple strategies to load related objects, and these strategies can be changed without rewriting the query or any other code. This makes it easy to experiment with different loading techniques to figure out which one is best for each particular use case. This technique was applied throughout the API to optimize the loading of related objects.

After the optimizations above, all API endpoints consistently responded in less than 100 ms for each request when navigating the web interface. While we did not collect further statistics on the API and database performance, we believe that it will be possible to solve similar performance issues in the future by using optimized database queries and potentially rewriting existing queries in another way to make use of indexes.

Client side performance

Since the heavy lifting of the user interface is done in the browser rather than on the server, the performance in HTML rendering is important for a good user experience.

Seemingly small bugs can introduce big performance regressions. One example is an event handler that was not properly unregistered when its element was removed. When new data arrived from the server, and the UI was updated, the event handler caused and reference to the old DOM tree to remain. This made it impossible for the garbage collector the free that memory. This caused the memory usage to increase with every update from the server, finally causing the browser to crash after a couple of minutes.

One of the best things with using AngularJS is that it keeps track of updating the UI automatically. The downside is however that it might lead to unnecessary rebuilds of some DOM elements when you are not being careful. Angulars ng-repeat directive suffers from this when replacing the data from the server. Angular the re-renders all the DOM-elements even if the new data from the server is still the same. By using the “track by” feature in ng-repeat, it is possible to help Angular to keep track of elements that already exists and does not need to be replaced.

To render a single view, the client often needs to make requests to multiple API endpoints and then combine the results into the final view presented to the user. We tried to minimize the number of requests by caching data in the browser that we consider to be static. CI types, and CI type relations are examples of API results that we request once from the server and then reuse during the same session.

Analysis & Discussion

As the project progressed, it became clear that there are a lot of other systems that needs access to this data via a structured API. Building a traditional web application where the server

generates the HTML would have forced us to build both an API and the HTML generator. The longer the project continued, we became more certain that a RESTful architecture was an overall good idea.

A RESTful architecture pushes some complexity from the server to the client. This approach would have been much harder a couple of years ago. Improvements in browsers (such as the HTML5 history API) and new JavaScript frameworks (such as AngularJS) make new things possible and allowed us to focus our efforts on a higher level. We think that this trend will continue, and that browsers and client side scripting will evolve and become more powerful. While some things are harder to achieve with pure client side scripting than traditional server side rendering, we still feel that this approach was the most useful in this case and provides a lot of benefits.

There were concerns about performance and scalability in order for Ericsson to continue developing this architecture and system. We have shown that it is possible to run the system efficiently on sparse resources and at the same time handling a lot of data. This is mainly the result of using existing and proven technologies such as PostgreSQL and SQLAlchemy. By leveraging the features of these tools makes it possible to add new features in a controlled way in the future. We did not have time conduct load tests to see how the system behaves under heavy load with a lot of simultaneous users, however we believe that this will not pose a significant problem. There are different techniques like further optimizing the code, adding load balancers and more servers that can counter such problems.

Being able to define types of CIs and at the same time having a timeline for all objects introduced by far the biggest part of the complexity in this project. We are not sure if that extra complexity in the database schema and in the code is worth to have the possibility to dynamically define new types. We had not time to create an admin UI to manage different types. This will probably be quite a time consuming task, since there needs to be API:s to interact with all type definitions in the database and UI to visualize it.

An alternative approach could be to define types as code level modules or plugins rather than putting all the type configuration in the database. This would allow for much more powerful customization of different CI types in the future.

Whether or not a plugin system like that is a good solution depends a lot on how quick and easy changes to the code can be deployed. In order to make this practically feasible there needs to be easy workflows in place that allows adding and modifying plugins and type definitions a almost daily basis.

Conclusion

In the end we saw that the application prototype with the database architecture fulfilled most of the requirements. The application is object oriented, handles dynamic types, performs well with a lot of data, handles changes over time in an elegant way and have a modern UI.

The way we approach the dynamic data types might not be the best way. We should have thought more about that in the beginning instead of getting stuck in the process of implement the solution in the database. In the end we may have ended up with a totally different and better solution.

Being able to do project management and creating test plans was a requirement from the start. During the project, we realized that the way test plans are managed are very different in different departments, even within the Linköping site. There are also a lot of other requirements regarding test setup planning, such as booking and reservations that looks very different from our original requirements.

If the aim for this system is to be used as a global tool to manage test environments for Ericsson, we believe that this system must stay very focused on only providing a Configuration Management database. We believe that the complexity of this system will be unmaintainable if features such as bookings, project planning, resource allocation and inventory management is introduced in the same system.

We rather believe that the solution for these problems are a natural extension of the RESTful architecture. REST APIs makes it possible to create multiple smaller and more focused systems that exchange information with eachother.

We had the opportunity to present our work and ideas to a lot of different people within Ericsson, where we got a lot of good feedback. Most of the feedback we received was very positive, and it is clear that this is an important and hard problem to solve in an organization that is as big as Ericsson.

Even if the system is not ready for production use we believe that this architecture overcomes the problems of the old system and shows a way forward. We hope that Ericsson will continue the development of this project and that our code and ideas will be useful in the future.

References

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
http://jacobian.org/writing/rest-worst-practices/

Responsiv design: http://liu.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:628224

Testramverk för distribuerade system:
http://liu.diva-portal.org/smash/record.jsf?searchId=1&pid=diva2:648970

Simplan: http://liu.diva-portal.org/smash/record.jsf?searchId=3&pid=diva2:601622

Sökfunktion: http://liu.diva-portal.org/smash/searchadthe.jsf

Todo:

Mer bakgrundsinfo så att man förstår lite mer om problemet
// Lagt till lite. Svårt att veta vad som saknas. Vad ville han ha mer av?

Någon referens till att SOAP är komplicerat
// Random länk mer om prestanda:
http://www2003.org/cdrom/papers/alternate/P872/p872-kohlhoff.html

Hur refererar vi till våra egna grejer? Vi kommer väl inte bifoga koden med rapporten

URL till worst practice rest api: http://jacobian.org/writing/rest-worst-practices/