POSEiDON

Pers**O**nalized **S**mart **E**nvironments to increase **I**nclusion of people with **DO**wn's sy**N**drome

Deliverable D5.4

Databases for integration of services

Call:	FP7-ICT-2013-10
Objective:	ICT-2013.5.3 ICT for smart and personalised inclusion
Contractual delivery date:	M6
Actual delivery date:	15.07.2016
Version:	V4
Author:	Lars Thomas Boye, Tellu AS
Contributors:	Dean Kramer, MU
Reviewers:	Silvia Rus, Fraunhofer Dr. Juan Carlos Augusto, MU
Dissemination level:	Public
Number of pages:	53

Contents

Executive summary

This deliverable documents APIs and data specifications for server-side data storage in the POSEIDON infrastructure. It is associated to Task 5.5 – Databases to support integration. This is version 4 of the deliverable, compiled together with version 4 of the framework deliverable D5.1. The server-side data storage has been defined as part of the framework, where it is an important aspect of the technical infrastructure. The main documentation of the infrastructure is found in D5.1. D5.4 is a compilation of specifications and documentation for application developers. The whole deliverable was rewritten for version 3, with mostly new content.

Version 4 update: The end user personalisation aspect has been clarified. See section 1.4 for an overview.

There are several key forms of data needed to be stored and shared between parts of the POSEIDON system. The SmartPlatform service holds the POSEIDON account with preferences, and also stores device data to be made available for monitoring. The data and access models of the SmartPlatform are documented in this deliverable, as are the API used to access the data.

A file server providing cloud storage of instructional content is also part of the infrastructure, with API documentation found here. Data specifications for the instructional content is part of the framework, and specifications for route data for navigation, products and shopping lists for money handling and a video play list are found in this deliverable.

Calendar data also needs cloud storage. The framework infrastructure includes Google's Calendar service for calendar event storage, but extends the event data with instruction lists and multi-media. Calendar documentation here covers architecture, data model and data specification for this part of the framework.

1 Introduction

1.1 Deliverable content and versions

The focus of this deliverable is the data storage needed to support the various modules of the POSEIDON system. The Description of Work calls for shared access databases which will act as interfaces amongst different parts of the system to support interoperability. Various forms of data needs to be stored in the system. User's data such as a profile with preferences, data for services, and context data for context awareness and monitoring are all important parts of the POSEIDON solution. All client applications should have access to the same data. Secondary and tertiary users use applications to enter personalized content and do other management tasks on the data. The same data must then be available for primary users, for training and preparations at home and at all times on the mobile device. Data must be connected to accounts, both to connect the right primary and secondary user and for secure authentication.

Data repositories and access is related to the development framework task as well as to the prototype functionality developed in the project. The nature of the data storage depends on the architecture and technology selected for the framework. The contents of this deliverable has changed significantly in the course of the project, based on the development of the prototype services and framework. Its planned date of delivery was month 6 of the project, along with the framework deliverable D5.1. First versions of the deliverables were delivered at this early stage of the project. However, it was not feasible to define a complete framework at this early stage, and the framework continued to evolve based on the development of the prototype system.

A second version was delivered in the second year, updated to reflect the state of the second iteration prototype system in the phase leading up to the first pilot. It had some documentation for SmartPlatform, considered as part of the framework from the start. And it had documentation for calendar and route data as it was used in that prototype iteration. While the SmartPlatform documentation were revised and extended, the other documentation was new for version 3. APIs and data specifications have been formalised and included as part of the POSEIDON framework. Everything in D5.4 is now considered part of the framework – it is documentation for developers to be able to use the data storage part of the infrastructure.

This third version of D5.4 was compiled together with the finalisation of the framework and its documentation in version 3 of D5.1. Version 4 of D5.1 and D5.4 are minor updates to better address personalisation (see section 1.4). D5.1 describes the development framework. The system architecture is described in chapter 4. The technical infrastructure, which includes the data stores, is described in chapter 5. These chapters should be read to get an overview of the framework and a conceptual understanding of the technical infrastructure. D5.4 compiles the technical specifications of APIs and data structures, needed for developers to connect to the infrastructure. Each chapter is based on a technical documentation for developers for one part of the infrastructure, and each of these documents is published on the POSEIDON project web[1] as part of the framework documentation for developers. Apart from the introduction, this text will have little interest to non-programmers, while D5.1 gives the more accessible descriptions.

1.2 Framework infrastructure

Figure 1 shows the technical architecture of the POSEIDON framework. The infrastructure is highlighted in colour, with different colours for different infrastructure components. This deliverable

[1] http://www.poseidon-project.org/

is concerned with the server-side data stores along the bottom of the figure, and especially the APIs of these data stores, as this is where client applications interact with the data stores. Such client applications are indicated in the upper part of the figure, and the lines with arrow-heads indicate some of the infrastructure usage done by applications. The interactive table and context middleware components of the infrastructure are not subjects of this deliverable, as these have no server-side API in the framework. The context awareness middleware has a server-side with data store, but this is private to the context middleware sub-system – applications interact with the middleware of the mobile device and not directly with the server.

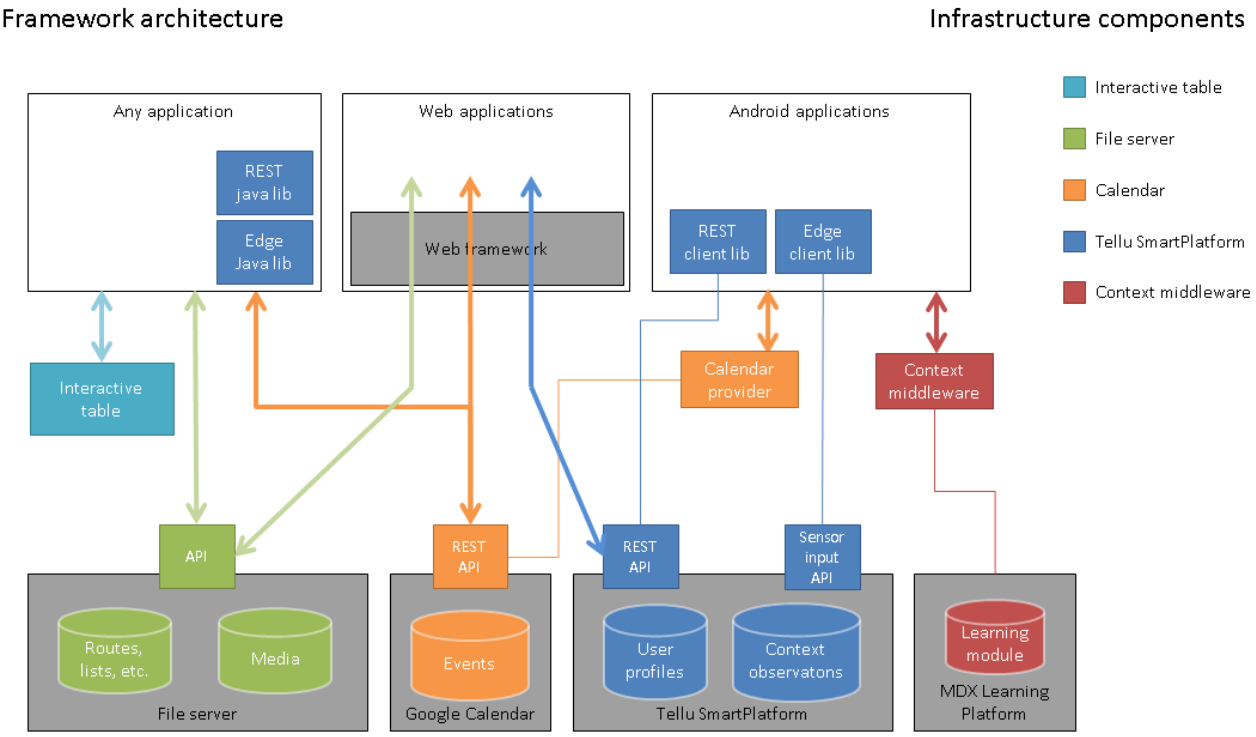

Figure 1: POSEIDON framework architecture with data stores

The SmartPlatform service (blue in Figure 1) stores both user accounts with preferences and observations such as position. It has a sensor input API for raw input of data, primarily used for position and other device data from mobile devices. Java and Android client libraries are provided to developers as part of the framework, handling the transactions on the most relevant platforms, so the API itself is not documented here. More important is the REST API, where applications get access to the SmartPlatform data. The developer documentation for POSEIDON usage of the SmartPlatform is found in chapter 2, and includes an overview of the SmartPlatform data model and related aspects, documentation of authentication and access and the API itself, and specification of POSEIDON-specific account properties. Detailed API documentation is included in an appendix.

The infrastructure includes a file server (green in Figure 1), providing a shared cloud storage of instructional material and other media. The file server API is documented in chapter 3. Data specifications for various forms of instructional content is also part of the infrastructure. Chapter 4 contains the data model and specification for route data, used to define routes for training and real-time navigation. Chapter 5 contains specifications for shopping list data for money handling functionality. Chapter 6 has a specification for a play list of videos.

Calendar data is stored in the Google Calendar service, but all user interaction with this data must be through POSEIDON applications as our data model is an extension of that used by other calendar services, including instructions and media. Access to the data is either through the Google Calendar HTTP API, or through the calendar data provider middleware on Android devices. The calendar architecture, data model, and data specification is documented in chapter 7.

1.3 Data security

All APIs described here use an encrypted HTTP connection (https) for transactions and require authentication. POSEIDON accounts are defined in Tellu SmartPlatform. These accounts are used for both SmartPlatform and file server transactions, with the file server connecting to the SmartPlatform instance for verification. In the simplified model of the pilot system, one account provides authentication for a secondary user and associated primary user. A user name and password must be provided in each application of the system. In primary user applications this can be entered by a secondary user as part of setup and thereafter remembered by the application. The account model of the SmartPlatform allows for a hierarchy of accounts, and it is possible to create tertiary user accounts with access to the data of multiple primary users. The account model is described in chapter 2.2

The calendar storage uses the Google Calendar service, and therefore requires authentication with a Google account. On Android devices this authentication is handled outside the POSEIDON system, as it is part of the device middleware. For other applications, the secondary user must give the application permission to access the calendar part of a Google account.

1.4 Personalisation

Personalisation is a very important aspect for the POSEIDON solution. Section 4.4 in deliverable D5.1 was added in the fourth iteration of the framework deliverables to clarify how the framework infrastructure supports personalisation. The main mechanisms are the user profile of the account, and the various forms of personal content. All parts of the infrastructure documented in D5.4 take part in this personalisation.

The user profile is stored on the SmartPlatform service, as properties on the asset entity. This is described in section 2.5. A subsection has been added to give a detailed explanation of how the profile is retrieved using the SmartPlatform API. Much of the personal content is stored on the file server. The API is described in chapter 3. The forms of personalised content defined by the framework are routes for navigation (data specification in chapter 4), shopping lists (chapter 5) and video lists (chapter 6). All of these include media, stored on the file server. Finally, calendar data is also a form of personalised content. The architecture and data model, described in chapter 7, is based on the Google Calendar service, but with a POSEIDON-specific extension described in section 7.4 for additional personalisation.

2 SmartPlatform

This chapter reproduces the POSEIDON developer documentation for the SmartPlatform framework component. It first describes the data flow and data and account model of the platform. It describes the API other POSEIDON components can use to connect to this service. And it describes what data we store in SmartPlatform in the POSEIDON prototype solution. An overview of the SmartPlatform is given in chapter 5 of deliverable D5.1.

Tellu SmartPlatform is a generic and highly configurable platform for data collection and processing. It is used to implement sensor-based services. The core functionalities are:

- Receival of data from a heterogeneous set of sensor devices and protocols.
- The storage of this data into an internal data model.
- Processing of this data by a rule engine.
- A web application for management and data access.
- A REST API to allow access to the data by other services.

This documentation is not a complete developer manual for the Tellu SmartPlatform, but aims to explain what is needed for developers of external applications to be able to connect to the platform. In addition to explaining the account and data model and the main API itself, it will explain aspects of the system that are useful as a background.

2.1 SmartPlatform data and interface overview

Figure 2 shows an overview of the platform.

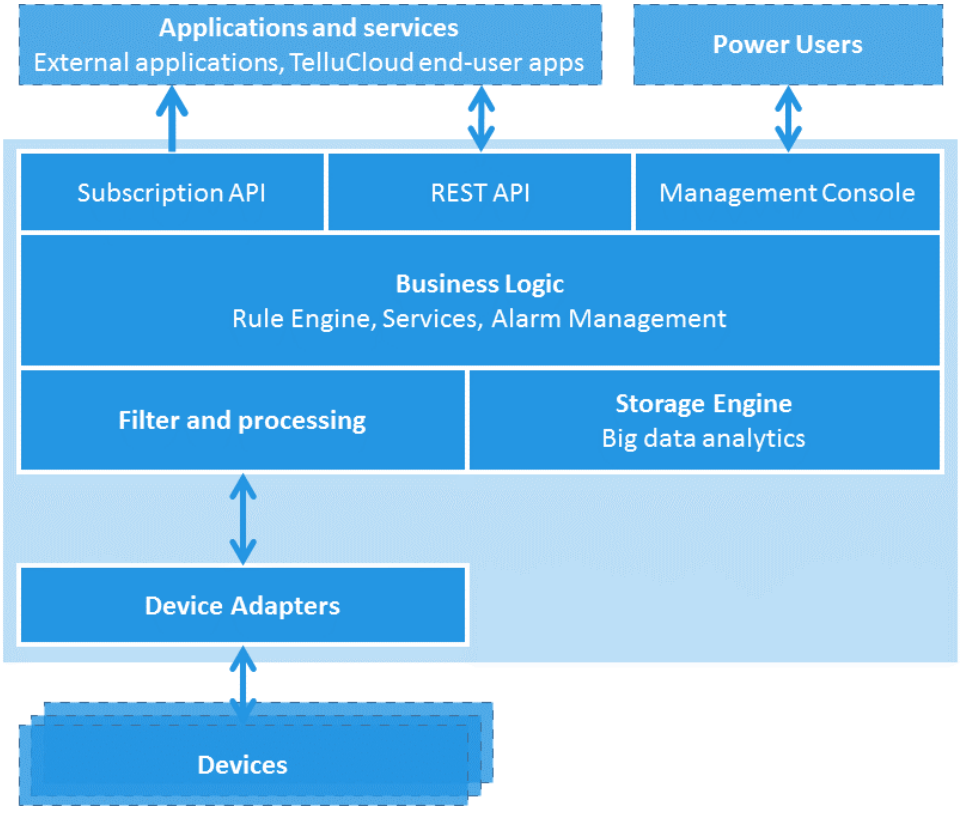

Figure 2: Tellu SmartPlatform architecture overview

There are two different interfaces to the SmartPlatform for other components in the POSEIDON solution – *Device Adapters* (edges) and *REST API* – and it is important to understand how these differ.

Roughly we can say that the first is for input of raw data in forms that are independent of SmartPlatform, while the second gives access to SmartPlatform's own data model, which includes results of the input after processing. So these interfaces are at opposite ends of the system. We will go through these aspects of the system, following the flow of data. But first a quick remark about "devices" and "sensor" data. We use these terms because the SmartPlatform was created for processing data from traditional sensor devices, with inputs such as position, temperature and events registered by the devices. However, as it is a generic platform, the data input can be anything that can be represented with numbers and text. So the "Sensor device" term should be taken in the broadest possible sense.

The SmartPlatform can be set up with Device Adaptors (often called edges) to collect data from sensor devices. The platform can be used with any type of data source by adding an edge that speaks its language and translates incoming data into the platform's internal format. In addition to receiving data from the devices and passing it on to the core, an edge typically supports commands to the devices, such as for configuration, so there is communication both ways. In POSEIDON we have so far only done data collection from client applications we develop within the project, where we have control of the communication protocol. We have a generic SmartPlatform edge where data can be posted using HTTP POST and a JSON format. This simple protocol is easy to implement in our components that needs to post data to the system. However, the edges give the system great flexibility. We can easily connect small purpose-built sensor devices such as a stand-alone GPS.

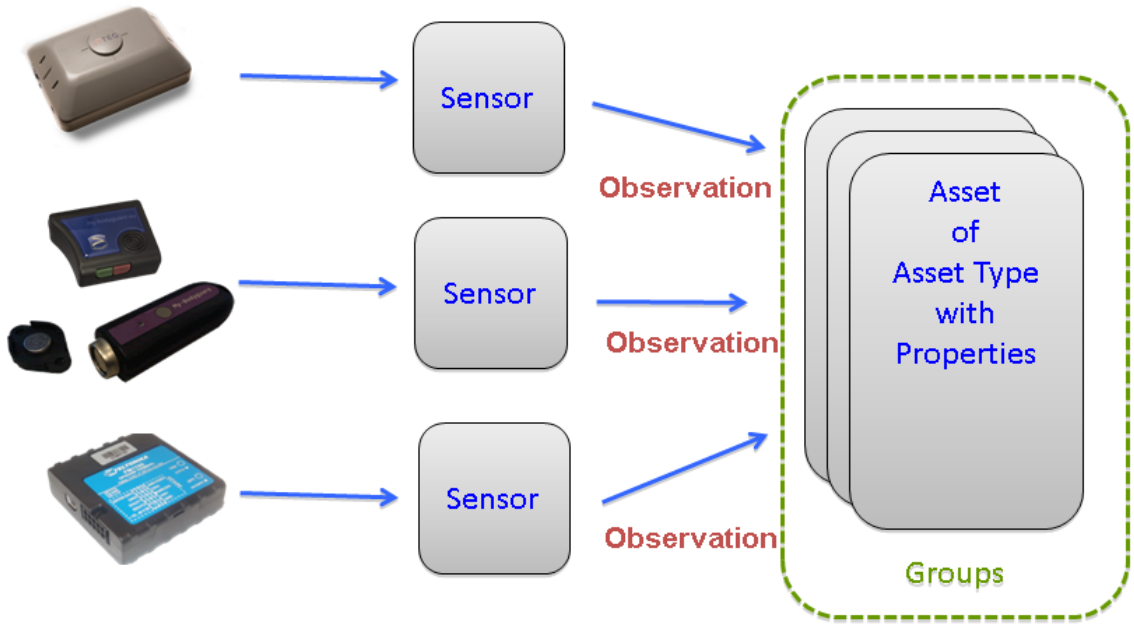

Figure 3: Asset and devices

As this edge part of the system is a sensor device interface meant for data collection, it is independent of SmartPlatform's internal data model. The relevant concepts in the data model are that of *device*, which is a source of data, and *observation*, which is an input to the system from a device. Observation is still called *position* in some places, as initially all observations were positioned, but in addition to some fixed fields such as timestamp and position data they can have arbitrary fields in a key-value map. The central concept in SmartPlatform is *asset*, which is a tracked entity. In POSEIDON, there is an asset representing each primary end user. An asset can have one or several devices, and observations from these devices are then known to be about this asset. An asset may for instance have a position, and it does not necessarily matter where the position comes from, so it

is possible to add a new device to provide position without affecting the service logic that deals with assets and their properties.

Figure 3 shows the relationship between sensor devices and an asset. It is basically the relationship between sources of data and the entity the data is about. The figure also indicates that assets can belong to groups, and that an asset can be of a specific *type*, with the type specifying properties all such assets have. Asset is also the main entity for POSEIDON use of the system. Using fields available for the asset entry, such as properties and tags, we can store context, preferences and other user data, and make it available both for the rule engine in SmartPlatform and for other modules through the API.

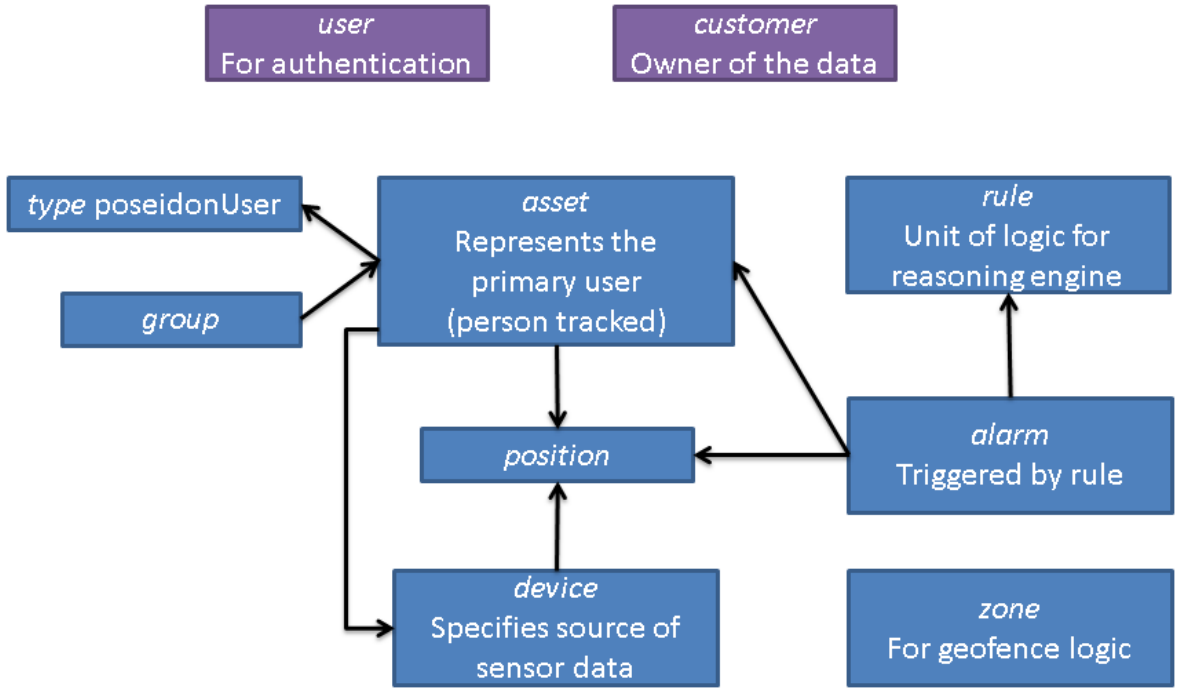

Figure 4: Central SmartPlatform data model entities

Figure 4 shows the most important entities in the SmartPlatform data model, and their relationships (references). Those not already described are mainly of interest to the platform and service itself, for the logic which can be built using its rule engine. An *alarm* entity can be created by the rule engine when some abnormality is detected. If this mechanism is used, alarm entities can be retrieved through the API. The *zone* entity is for creating geofence logic in SmartPlatform, that is, trigging rules based on entering and leaving geographical areas. This can be used to keep track of where a user is (home/school/etc.).

The data model with entities such as assets is stored in SmartPlatform's *Storage Engine* and can be managed through its *Management Console*. Data from edges go through *Filter and processing*, where unneeded data are discarded and the rest is connected to the data model. Every time there is a new observation available from the initial processing, the *Business Logic* (rule engine) processes the change in state, and this can cause rules to trigger, which in turn can update asset state.

The REST API gives access to the objects of SmartPlatform's data model, both for reading and for making changes. The asset acts as a repository of information about the user, and will hold context

such as location, user preferences, and any other types of data that needs to be shared between POSEIDON modules. Fields can have their values updated directly from observations, from the triggering of rules, or through the REST API. Generally, if information may need to trigger rules in SmartPlatform's rule engine, it should be posted as an observation through an edge, otherwise it can be posted directly to the asset through the REST API.

The SmartPlatform also has a *Subscription API*, where external applications can register subscriptions, for instance to all updates pertaining to a specific asset. The platform then maintains an active connection to the subscribing client, rather than requiring the client to poll for data to check for updates. The POSEIDON web application uses this to show the primary user position on a map.

More of the entities of the data model are described in the appendix, in the context of the REST API.

2.2 Accounts and authentication

SmartPlatform has an extensive, hierarchical access control scheme, which is described in this chapter.

2.2.1 API access

All access to the system through the REST API requires authentication as a *user* with the right permissions. A user in the SmartPlatform context is someone (or an external system) with access to the system, registered in the system with a user name and password (not to be confused with users in the POSEIDON meaning, although a POSEIDON secondary user may have a SmartPlatform user).

Each API request must include an authentication token, supplied as an HTTP header, which is tied to a user. A token can be acquired in a login transaction giving a user name and password. A token may be time-limited or not. For the other modules in the POSEIDON system, it is possible to issue tokens with no timeout to simplify their interaction with the service.

2.2.2 Account model

Figure 5 shows the main entity types relevant to this data access, where it is important to understand the distinction between account, user and the tracked POSEIDON primary user. At the top of this hierarchy there is a *service provider*, which can be the administrator of a set of accounts. A POSEIDON service provider has been set up for the POSEIDON prototypes. All data available through the REST API is owned by an *account;* this entity type is called *customer* internally and in APIs. Then we have the user, which is just a data access concept as has already been discussed. A user is always tied to a specific account, and can only access the data in this account. But an account can have many users. A user can also have access to the higher levels of the hierarchy. It can be a *service owner*, which means it can manage service providers, or it can be a service provider. A service provider user has access to all the accounts of this service, and can change which account it is accessing through the API.

In addition, there is a very detailed system of permissions which specify exactly which entity types, and which operations on this data, a user has access to. These are collectively managed as *roles*. For instance, an account may have one user with an administrator role, with full permissions to configure the account, and other users with a much more restrictive role to just look at the data.

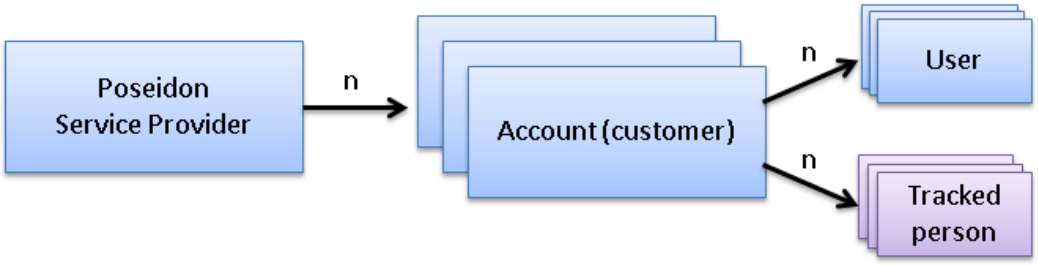

Figure 5: SmartPlatform account model

The POSEIDON primary users are represented as asset entities in the data model, as has already been mentioned. So such a tracked person is “just data” in this context, one of the data objects belonging to an account, and there can be any number of these stored in an account. It is possible to restrict a user’s access to specific assets in an account. All this means it is possible to use a single account for several sets of secondary users each having access only to a specific primary user.

When using the SmartPlatform as part of the infrastructure providing a service to primary, secondary and tertiary users, an account and role policy must be specified. A service provider responsible for this part of the service may set up a separate account for each primary/secondary user pair, or put all users in a single account. If correctly set up with permissions, the end users will not see any difference. The single account model has some important advantages. It makes it easy to give access to the data of a set of primary users, for instance to tertiary users. It also makes it possible to design service logic for the rule engine which considers multiple users, as all rules are account-specific. There is also more work to manage multiple accounts, as entities such as asset type and properties are also all account-specific and must then be duplicated between accounts. For the POSEIDON prototypes, we use a single account for all pilot users.

2.2.3 Device ID

The account and authentication scheme described so far does not concern the Device Adapter part of the system. Each data source (sensor device) has a unique ID in SmartPlatform, and this is what identifies observations as belonging to a specific device entity, which in turn belongs to an asset. A client application instance which will post data through an edge needs to be registered in the SmartPlatform instance, and it needs to know its ID so its posted data will be handled by the system. There are two ways to handle this. One way is to use the device command system supported by edges, where a configuration command is delivered to the device or application instance through the protocol implemented by the edge. The other is to use the REST API, knowing the name of the asset representing the end user, we can retrieve the device IDs for any devices registered for this user.

2.3 REST API

The SmartPlatform API is a standard HTTP REST API, supporting GET, POST, PUT and DELETE operations. Data is exchanged in JSON[2] format. This chapter gives a technical description of the format of API requests and replies. The relevant resources (data objects) are described in the appendix.

Note that the documentation for the API is available online at the following URL:

[2] http://www.json.org/

https://telludoc.atlassian.net/wiki/display/SMARTTRACKER/Smarttracker+API+v3

Refer to this for the latest version, and for examples of the JSON data objects.

2.3.1 URL

The URL consists of four parts: base (server address), customer, resource and ID.

```
<base url>/<customer id>/<resource>/<resource id>
```

Retrieving only the root of the URL (without resource) will give an object describing what resources are available.https://telludoc.atlassian.net/wiki/display/SMARTTRACKER/Smarttracker+API+v3

Property	Description
providers	A list of service providers the user making the request can access.
customers	A list of customers the user making the request can access.
access	A list of available resources with a map per resource indicating what methods that is available and whether the client is allowed to perform them.
features	A list of suggestions to the client to enable or disable features in order to provide a simpler interface to the user.
user	An object containing the id of the user making the request.
provider	An object containing the id of the service provider of the customer in the request.
customer	An object containing the id of the customer in the request (or if the customer id was not included in the URL, the customer associated with the user).
time	The time the request was handled.

2.3.2 Retrieving data

All data requests must be done with the HTTP method GET. All requests done on resources will have the same properties in the response.

Property	Description
result	A list of resources matching the data request. This will always be a list, even if the client requests a resource with a specific id.
total	The total number of resources matching the data request.
offset	Marker to use to paginate the data.
max	The maximum number of resources in each response.
user	An object containing the id of the user making the request.

provider	An object containing the id of the service provider of the customer in the request.
customer	The customer used as source of data in the request.
time	The time the request was handled.

2.3.3 Filtering data requests

The SmartPlatform has a powerful filtering mechanism. Filters are added as parameters in the URL. Multiple filters can be added, but a mechanism can only be used once per property (latitude:less=59 and latitude:greater=58 is possible, but name:contains=e and name:contains=m is not). All filters follow the same pattern.

```
<property name>:<filtering mechanism>=<filter value>
```

Filtering mechanisms	Description
equals	Usable on most data types.
contains	Usable on string data types and some more complex types.
less	Usable on number and date data types.
greater	Usable on number and date data types.

Example	Limit request to resources with ...
name:equals=Demo	name equal to Demo.
name:contains=em	Name containing the text “em”.
latitude:less=59	latitude less than 59.
longitude:greater=11	longitude greater than 11.
timestamp:greater=2013-04-18T00:00:00.0	timestamp after April 18. 2013.
timestamp:less=2013-04-20T00:00:00.0	timestamp before April 20. 2013.

2.3.4 Data content

When requesting data, not all data is included due to performance and bandwidth reasons. When querying a list of data, only id and name is included by default. When querying a single resource, all immediate properties are included (without any recursion). Complex objects will (usually) include an id and name. This behaviour can be overridden by adding a parameter to the URL named select. Select accepts a list of property names separated by the character “+”. It also has two special values, star “*” and at “@”. The symbol “*” includes all properties and all subproperties. The symbol “@” includes all properties but only the minimum of subproperties (id and name).

Example	

select=*	All properties of the resource, and all subproperties
select=@	All properties of the resource, but minimum of data for subproperties
select=lastValidPosition+type	Only lastValidPosition and type properties. Type (a complex type) will only have id and name.
select=lastValidPosition+type.icon	Only lastValidPosition and type properties. Type will now also have icon as well as id and name.
select=type.*	Only type. All properties of type will be included.
select=positionProvider.@	Only positionProvider. The immediate properties of positionProvider is included.

2.3.5 Submitting data

Adding a resource must be done with HTTP method POST, without a resource ID in the object or in the URL. Resource objects are wrapped in a list to allow creating more than one object in the same request.

POST `<base>/<customer id>/<resource>`

Updating an object must be done with HTTP method PUT, with a resource ID in the URL. In both cases the resource must a JSON object inside a JSON list in the request payload. See each resource section for more information about which properties that are required and valid values. The resource object is wrapped in a list to be consistent with creating an object. If a property is omitted then it will not be changed on the server.

PUT `<base>/<customer id>/<resource>/<resource id>`

2.3.6 Deleting data

Deleting data must be done with HTTP method DELETE with a resource ID in the URL. The response if successful is an empty GET response (with HTTP code 200).

DELETE `<base>/<customer id>/<resource>/<resource id>`

2.4 Tracking and logging history

This section describes the control of storage of tracked data history in the SmartPlatform service, and how the data can be accessed. This is done through the management console, by a developer, researcher or other personnel with an administrator account in the system.

The *Devices* category in the SmartPlatform account shows the last observation that has been made. *Accumulated Observation Values* returns the latest value for various properties. In the *Device* view, the *Debugging Tools* button shows a list of the latest observations, the list can be expanded when blue 'i' icon is clicked.

When the *Personnel* item in the *Content* menu is clicked, the asset properties (with current values) as well as the last observation from the associated device is shown. There is also a button for the History function. When tracking of the person is enabled, all asset property changes are stored, and will be available in this history view, as a table, graph or on a map, depending on the type of data.

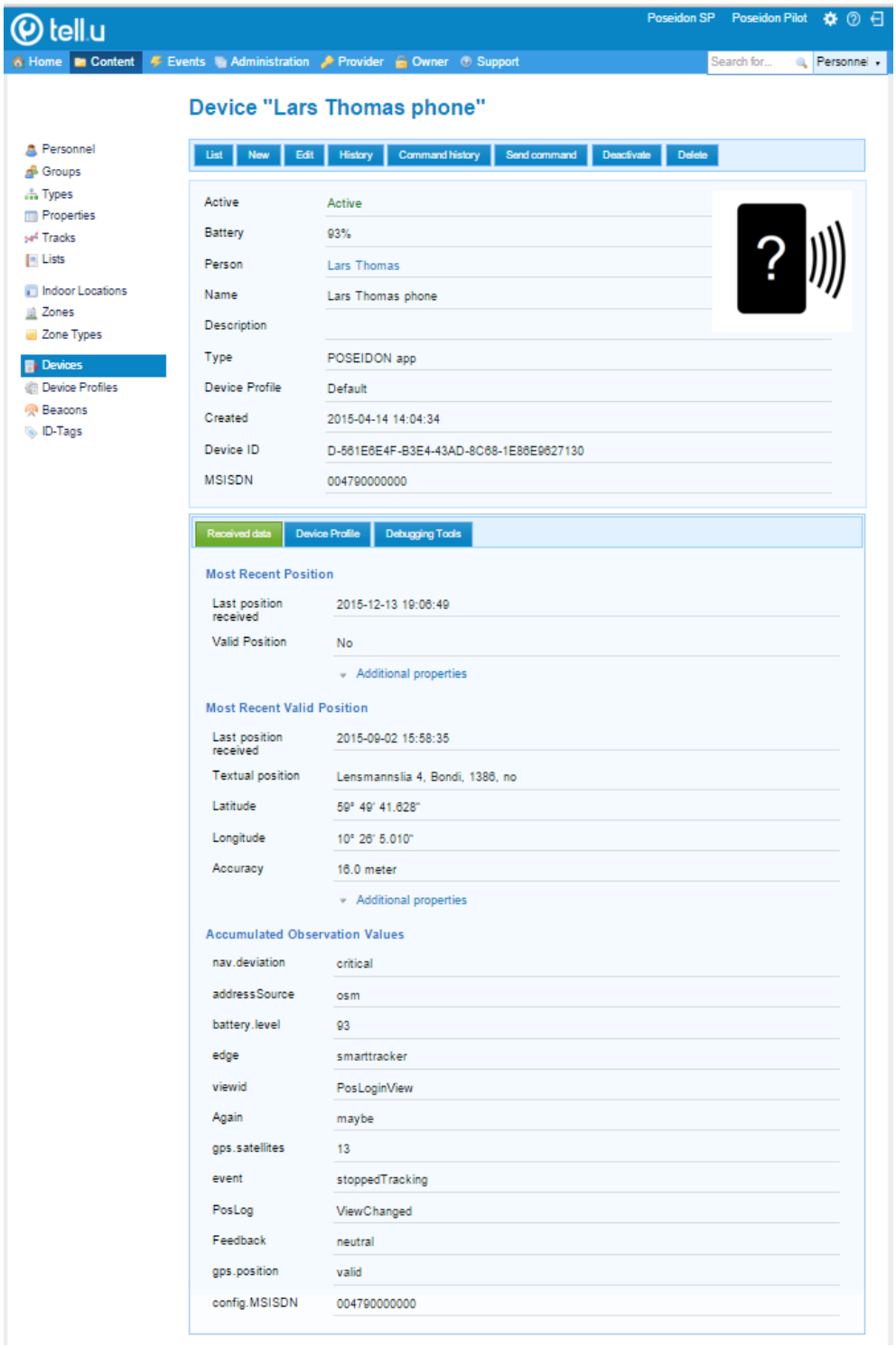

Figure 6: Device view in management console

Data is loaded for a specific timeframe. Tracking, in the sense of storing historical data, can be on or not, and how long the data is stored is configurable. It is not needed to simply keep track of where a person is at the current time. It is needed if a history of movement may be wanted. It also allows using the SmartPlatform service to log events from client applications. In the POSEIDON pilots the mobile application logged end user events to the SmartPlatform server, so that researchers could analyse the logs after the pilots, seeing how much various functions were used. Therefore tracking of history was enabled.

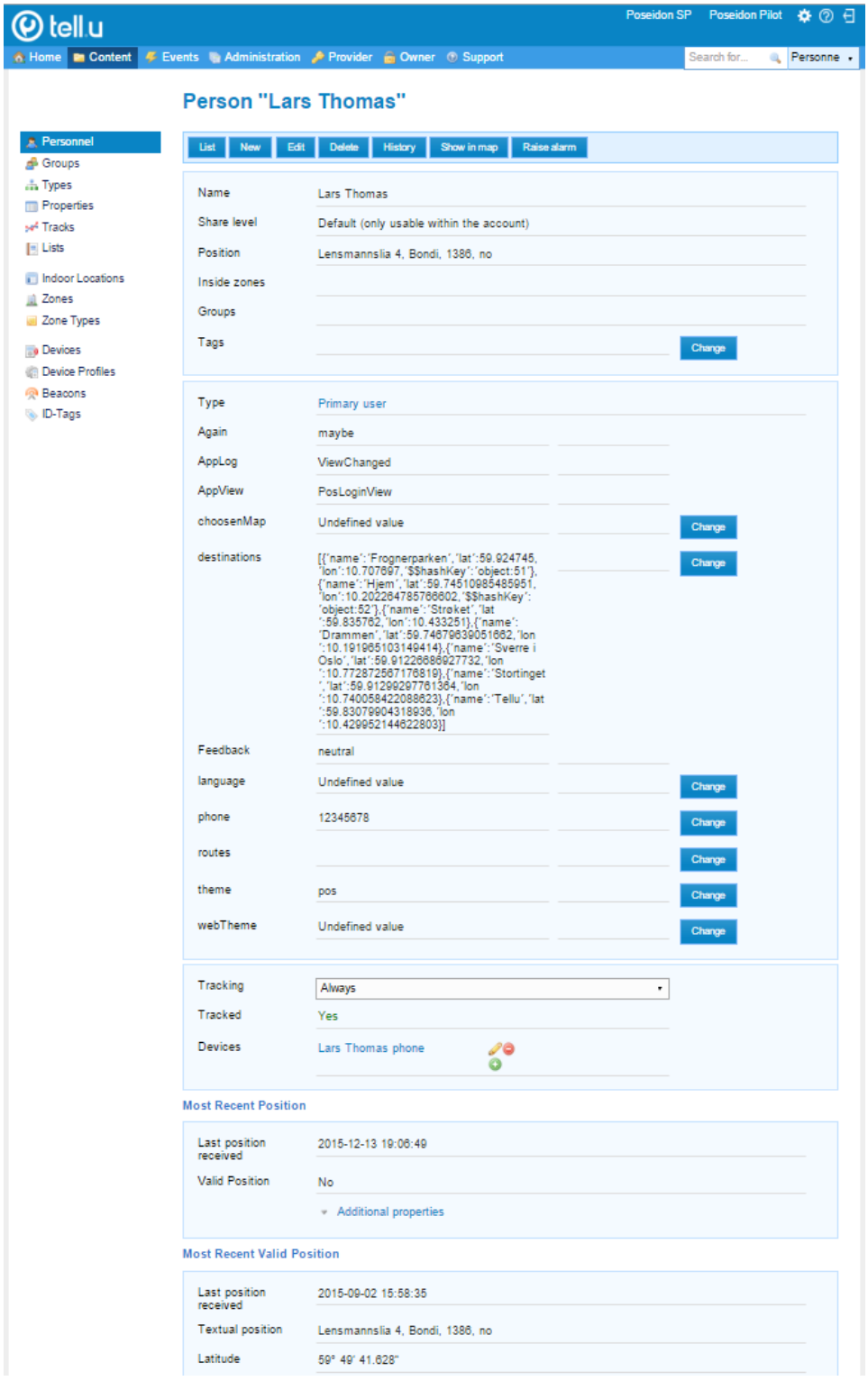

Figure 7: Asset view in management console

Tracking history storage can be affected by one of the following settings:

- The service provider level specifies default and maximum history lengths for accounts belonging to this provider. In the POSEIDON prototype this length is 90 days.
- Any account can set a history length within the service provider limit mentioned before. Thus, we sat the history length to 90 days in the pilot account (POSEIDON Pilot).

- History is only stored when *Tracked* property of the individual asset (tracked person) is enabled. However, tracking can be toggled through the management console, through a rule or through the REST API. *Tracking* option can also be set to *Never* or *Always*.

All of the above are related to storage of data on the server, and should not be confused with the tracking setting in the user preferences in the POSEIDON prototype mobile application. When this is turned off, device tracking is not sent to the server in the first place.

2.5 POSEIDON asset type and properties

The asset representing the POSEIDON primary user in the SmartPlatform service needs to be of an *asset type* defining a set of properties. These properties makes up a user profile. This is a place for applications to store shared preferences, such as for personalisation. An asset property is also needed for observation data to be stored in asset history.

The POSEIDON asset type is called “Primary user”. Figure 8 shows the properties defined at the time of the second pilot, in the management console. Some of these are used by specific applications to store their own preferences, and so not part of the framework.

Properties

Add Remove

Property	Name	Property type		Value type	Default Value	Unit	
Feedback	Again	From device	Again	Text	-	-	
AppLog	AppLog	From device	PosLog	Text	-	-	
AppView	AppView	From device	viewid	Text	-	-	
Feedback	Feedback	From device	Feedback	Text	-	-	
alarmInsistent	alarmInsistent	User input		Text	false	-	
alarmOnStart	alarmOnStart	User input		Text	true	-	
choosenMap	choosenMap	User input		Text	osm	-	
destinations	destinations	User input		Text	-	-	
language	language	User input		Text	en	-	
mobileStreaming	mobileStreaming	User input		Text	true	-	
phone	phone	User input		Text	-	-	
posInterval	posInterval	User input		Numeric	600	-	
routes	routes	User input		Text	-	-	
theme	theme	User input		Text	pos	-	
webTheme	webTheme	User input		Text	default	-	

Figure 8: Asset properties in POSEIDON pilots

Those properties which have an interest outside of a specific application are described in the table below. The framework allows extending the type with more properties as needed.

Property name	Description
destinations	A set of destinations the primary user may want navigation assistance traveling to. Used by the mobile application to create a new route to a

	wanted destination. It is stored as a JSON list, where each element has the properties name, lat and lon (the two last are the coordinates).
language	Language preference, currently used by the web application.
phone	The phone number of the carer to contact if the primary user needs assistance.
theme	The visual theme for the primary user. Currently supported are “pos” and “posHC”, for the default POSEIDON theme and a high-contrast alternative.
webTheme	Visual theme for the secondary user (used by the web).
Feedback	For logging from the mobile application – feedback on the use of app functionality.
Again	For logging from the mobile application – whether the user would like to use the functionality again.
AppLog	For logging from the mobile application – application event.
AppView	For logging from the mobile application – a view is shown in the app.

2.5.1 API usage for user profile

As the properties make up the user profile shared by POSEIDON applications, retrieving the asset with properties is the most important interaction with the SmartPlatform service for POSEIDON applications. We therefore show the details of the API calls here.

First we need to log in, to get an authentication token:

URL	`https://ri.smarttracker.no/web/api2/login`
Method	`POST`
Headers	`Content-Type: application/json`
Payload example	`{"username":"example@poseidon.no", "password":"example123"}`

This returns a JSON with a token:

```
{
	"token": "12345678-1234-1234-1234-1234567890ab"
}
```

Next we retrieve account information, to get the customer ID for subsequent calls. The token must be included as a header in this and all subsequent calls:

URL	`https://ri.smarttracker.no/web/api3/`
Method	`GET`
Headers	`X-Auth-Token: <TOKEN>`

The account information has many fields, but we are interested in the customer ID:

```
{
	...
	"customer":{
		"name": "Poseidon Pilot"
		"id": 12345
		...
	}
	...
}
```

Next we ask for assets of the customer. This is always a list, but in POSEIDON it should only be one, which is the primary user.

URL	https://ri.smarttracker.no/web/api3/<CUSTOMERID>/asset
Method	GET
Headers	X-Auth-Token: <TOKEN>

We need the ID of the asset from this JSON:

```
{
	"result":[
		{
			"name": "Some Name"
			"id": 67890
			...
		}
	]
	...
}
```

Finally we can retrieve the asset details based on the ID.

URL	https://ri.smarttracker.no/web/api3/<CUSTOMERID>/asset/<ASSETID>
Method	GET
Headers	X-Auth-Token: <TOKEN>

In this reply, we are interested in the list of properties:

```
{
	"result":[
		{
			"name": "Some Name"
			"id": 67890
			...
			"properties":[
			{
				"name": "mobileStreaming"
				"timestamp": "2016-04-27T07:42:28.669"
				"value": "true"
				"numberValue": null
				"type": "value"
				"valueType": "text"
				"unit": null
				"typePropertyIdentifier": "5dc15be4-9e1f-4366-bace-
			6b5f07c4ca31"
				"propertyTypeIdentifier": "f4597f3c-2d6a-43b2-872d-
			36f7237867b7"
			},
```

```
                {
                        "name": "alarmOnStart"
                        "timestamp": "2016-04-27T07:42:07.685"
                        "value": "true"
                        "numberValue": null
                        "type": "value"
                        "valueType": "text"
                        "unit": null
                        "typePropertyIdentifier": "c0b8bc97-acd1-4fe0-b925-
                0d486dd507b1"
                        "propertyTypeIdentifier": "da852545-7ab8-412b-abd1-
                ccdce52738d0"
                },
                {
                        "name": "alarmInsistent"
                        "timestamp": "2016-04-27T07:43:59.31"
                        "value": "false"
                        "numberValue": null
                        "type": "value"
                        "valueType": "text"
                        "unit": null
                        "typePropertyIdentifier": "affbaae9-4d4f-4b81-a9df-
                        79c30092a2f8"
                        "propertyTypeIdentifier": "d0f5d889-c5e8-48e3-b543-
                7a8ce90d22b8"
                },
                ...
                ]
                ...
        }
    ]
}
```


The example JSON only shows the first three properties – there will be many more. For reading, we are interested in the name and value fields.

If an application allows the user to change property values, it should post the changes back to SmartPlatform. It should do a PUT transaction with the same URL (with asset ID), with a JSON body including the changed properties.

3 File server API

This chapter gives an overview of how the file-server of the POSEIDON infrastructure works, and a description of how to use it.

The file server may be deployed at any server, but for the POSEIDON prototypes it is deployed on the SmartTracker research and innovation server where the POSEIDON web application is deployed. This may change in the future. The term *<BASE_URL>* in the documentation will therefore be referring to *https://ri.smarttracker.no*.

3.1 Logging in

To be able to use the file server the client has to authenticate with the authentication token identifying the user in SmartTracker. This token can be received either by logging in using the SmartTracker API directly or by logging in using the file server API.

3.1.1 Request

Method	POST	<BASE_URL>/files/login.php
Headers	Content-Type	application/json

Payload	JSON-Object	{
		”username”:”<USERNAME>”,
		”password”:”<PASSWORD>”
		}

<USERNAME> and <PASSWORD> is you username and password in SmartTracker.

3.1.2 Returns

On success the request above will return a JSON-object containing the authentication token that is used for interactions with the file server.

```
{
    token: "36346d0f-6d08-423e-8059-491b1144ab6f"
}
```

3.1.3 Errors

CODE	Error	Description
402	Payment Required	If authentication data is wrong
400	Bad Request	If body is missing or malformed

3.2 Adding a resource

A file resource is added by posting a binary file and a category to the file server. The file will be given an UUID, and will be stored in a category.

3.2.1 Request

Method	POST	<BASE_URL>/files/resource.php
Headers	Content-Type	multipart/form-data
	X-Auth-Token	<TOKEN>
Payload	type (TEXT)	String value containing the category of the resource.
	file (FILE)	The binary file that should be uploaded
	assetID (TEXT)	(OPTIONAL) indicating which asset the resource should be associated with. This only applies to users that control multiple assets.

3.2.2 Returns

If successful, the request should return a JSON object containing the UUID of the resource.

```
{
	success: true,
	message: "File successfully uploaded",
	resourceID: "9db2c2fe-da91-429d-bb50-63e512a93328"
}
```

3.2.3 Errors

Code	Error	Cause
400	Bad request	If the authentication token is not provided
401	Unauthorized	If the authentication token is not valid
400	Bad Request	If *type* is not provided
400	Bad Request	If *file* is not provided
500	Internal Server Error	Should not happen, but still…
400	Bad Request	If the user controls multiple assets and *assetID* is not provided
401	Unauthorized	If the *assetID* provided is not controled by the user authenticated.

3.3 Changing a resource

There are two things that can be changed for a resource: Either the category of the file or the binary content of the file resource. These operations cannot be executed simultaneously. If you want to change them both, you have to do it in two operations. If both the file content and the category are provided in the request, only the file content will be changed.

3.3.1 Request

METHOD	POST	**<BASE_URL>/files/resource.php**
HEADERS	Content-Type	multipart/form-data
	X-Auth-Token	<TOKEN>
Payload	resourceID (TEXT)	The UUID of the resource that should be altered.
	file (FILE)	(OPTIONAL) Binary file with the new content. Type will not be changed if *file* is provided.
	type (TEXT)	(OPTIONAL) Category of the file. Will be ignored if *file* is provided.

	assetID (TEXT)	(OPTIONAL) indicating which asset the resource should be associated with. This only applies to users that control multiple assets.

3.3.2 Returns

If successful the request will return a json object describing the change. Either:

```
{
	success: true,
	message: "New content for resource: '9db2c2fe-da91-429d-bb50-
	63e512a93328'"
}
```

when changing the file content or when changing the category:

```
{
	success: true,
	message: "Type has been changed for the resource: '9db2c2fe-da91-429d-
	bb50-63e512a93328'"
}
```

3.3.3 Errors

Code	Error	Cause
400	Bad request	If the authentication token is not provided
401	Unauthorized	If the authentication token is not valid
404	Not found	If the no file with the provided *resourceID* can be found.
500	Internal Server Error	Should not happen, but still…
400	Bad Request	If the user controls multiple assets and *assetID* is not provided
401	Unauthorized	If the *assetID* provided is not controlled by the user authenticated.

3.4 Listing resources

Resources can be listed by category, or as a complete list of resources for an asset. The entries in the list will contain the original name of the file resource, the UUID of the resource, the file's mime type and a md5 hash checksum of the file.

3.4.1 Request

METHOD	GET	**<BASE_URL>/files/resource.php**
Headers	X-Auth-Token	<TOKEN>
Parameters	type (TEXT)	(OPTIONAL) Category of the file. Will list only resources in this category if provided.
	assetID (TEXT)	(OPTIONAL) indicating which asset the resource should be associated with. This only applies to users that control multiple assets.

Example: GET https://ri.smarttracker.no/files/resource.php?type=images

3.4.2 Returns

Successfull result will return a JSON array containing metadata for the requested resources.


```
{
    image: [
    {
        resourceID: "0408c04f-8ac9-421e-bf98-9fe595c404ef",
        name: "Trondheim.jpg",
        mime: "jpg",
        md5: "8876bc2339fd622cd99ca4bbf2651613"
    },
    {resourceID: "90db5641-c74c-4a4c-b13d-
    ad20623567b7", name: "Snøkanon.jpg", mime: "jpg",…},
    {resourceID: "b351a3be-1f22-4267-9bc7-
    57cb65d427d2", name: "Trondheim.jpg", mime: "jpg",…}
    ],
    route: [
    {resourceID: "9db2c2fe-da91-429d-bb50-
    63e512a93328", name: "directions.json", mime: "json",…},
    {resourceID: "ba19decd-ebbd-45b3-afdb-
    56806db96bcb", name: "directions.json", mime: "json",…},
    {resourceID: "c8e12b5d-0f76-463f-9470-
    50ca45d4b066", name: "directions.json", mime: "json",…}
    ],
    shoppinglist: [
    {resourceID: "150c6ccf-2ea8-4a6c-a2ea-
    173002374155", name: "shoppinglist.json", mime: "json",…},
    {resourceID: "1dc5b4ea-7cd6-449e-ac7d-
    527380063ac2", name: "directions.json", mime: "json",…},
    {resourceID: "1df79998-dc93-483e-8ffa-
    34d739b394c6", name: "shoppinglist.json", mime: "json",…},
    {resourceID: "bf034f0f-8333-435a-8342-
    40894c504b3c", name: "shoppinglist.json", mime: "json",…}
    ]
}
```


or by category (image):


```
[
    {
        resourceID: "0408c04f-8ac9-421e-bf98-9fe595c404ef",
```

```
        name: "Trondheim.jpg",
        mime: "jpg",
        md5: "8876bc2339fd622cd99ca4bbf2651613"
        },
        {
        resourceID: "90db5641-c74c-4a4c-b13d-ad20623567b7",
        name: "Snøkanon.jpg",
        mime: "jpg",
        md5: "1358c53f9572424ba30565d8147a2937"
        },
        {
        resourceID: "b351a3be-1f22-4267-9bc7-57cb65d427d2",
        name: "Trondheim.jpg",
        mime: "jpg",
        md5: "8876bc2339fd622cd99ca4bbf2651613"
        }
]
```


3.4.3 Errors

Code	Error	Cause
400	Bad request	If the authentication token is not provided
401	Unauthorized	If the authentication token is not valid
500	Internal Server Error	Should not happen, but still…
400	Bad Request	If the user controls multiple assets and *assetID* is not provided
401	Unauthorized	If the *assetID* provided is not controlled by the user authenticated.

3.5 Fetching a resource

The resource is fetched by a GET request that returns the binary file.

3.5.1 Request

METHOD	GET	**<BASE_URL>/files/resource.php**
Headers	X-Auth-Token	<TOKEN>
Parameters	resourceID (TEXT)	The UUID of the resource to be fetched
	assetID (TEXT)	(OPTIONAL) indicating which asset the resource should be associated with. This only applies to users that control multiple assets.

3.5.2 Returns

A successful result will return the file as binary data, with the file's original name.

3.5.3 Errors

Code	Error	Cause
400	Bad request	If the authentication token is not provided
401	Unauthorized	If the authentication token is not valid
404	Not found	If the no file with the provided *resourceID* can be found.
500	Internal Server Error	Hopefully you won't experience this one. Notify developer if you do.
400	Bad Request	If the user controls multiple assets and *assetID* is not provided
401	Unauthorized	If the *assetID* provided is not controlled by the user authenticated.

3.6 Deleting a resource

Deleting a resource will remove it permanently from the file server.

3.6.1 Request

METHOD	DELETE	**<BASE_URL>/files/resource.php**
Headers	X-Auth-Token	<TOKEN>
Parameters	resourceID (TEXT)	The UUID of the resource to be deleted
	assetID (TEXT)	(OPTIONAL) indicating which asset the resource should be associated with. This only applies to users that control multiple assets.

3.6.2 Returns

A successful result will return confirmation that the resource has been deleted.

```
{
      success: true,
      message: "The resource with id '0a542d5b-29a9-4568-b8fe-3a2b4b9a2ad3'
      was successfully deleted"
}
```

3.6.3 Errors

Code	Error	Cause
400	Bad request	If the authentication token is not provided
401	Unauthorized	If the authentication token is not valid
404	Not found	If the no file with the provided *resourceID* can be found.
500	Internal Server Error	Hopefully you won't experience this one. Notify developer if you do.

400	Bad Request	If the user controls multiple assets and *assetID* is not provided
401	Unauthorized	If the *assetID* provided is not controlled by the user authenticated.

4 Route data specification

This chapter reproduces the POSEIDON specification of route data for developers. A route specifies how to get from a start location to a destination. It needs to provide the instructions necessary to guide the person with Down's syndrome, and these typically need to be personalised by a carer. Route data can be used to present the route, in rehearsing the trip, and in real-time navigation guidance and tracking. The route data is stored as structured data in a JSON file, plus associated media files for the instructions. In the POSEIDON prototype system, these files are stored on the POSEIDON file server and accessed by all the applications (stationary, web and mobile).

4.1 Data model

As a basis for the route data, we have a generic conceptual data model, to capture the essence of what is produced by typical route planners and needed for route presentation and navigation. This model is based on route data formats of major route planning services, specifically OpenTripPlanner[3] and Google Directions[4]. The routes produced by these two systems are similar on the conceptual level, although they differ in some details. Adopting a similar model is done both because it is a proven approach, and because it facilitates interoperability, making it possible to use the same navigation algorithms etc. for personalised POSEIDON routes and routes from the route planning services. It is a hierarchical model, shown in Figure 9.

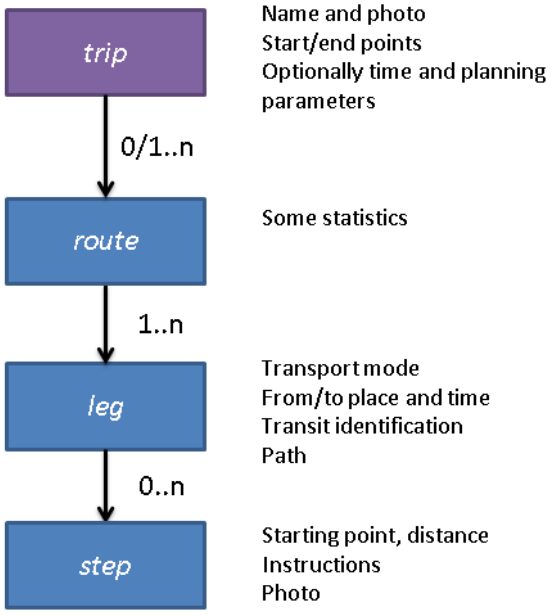

Figure 9: Route data model

The top-level element of our model is *trip*, which is a header or abstract definition for route data, as opposed to the route itself. It can represent meta-data for a planned route, or input parameters for route planning. As a minimum, it specifies a destination point, so that it can be used to get a route from wherever the person is currently located from an automated route planning service. It can also specify a start point, a time for the start or destination, preferred transport modes, and other parameters for route planning. A list of preferred routes or destinations for the user to choose from is a list of *trip* elements. For rehearsal or navigation, a *route* for the trip is necessary. There can be several *routes* for a trip in some cases – a route planning service may provide several alternative routes, and they can be processed until one is chosen for navigation.

The *route* element represents an actual route – the way from a starting point to a destination. The *route* element itself is mainly a container for a set of *legs*, though it can also have some overall statistics. The route is made up of *legs*, which is a way to partition the route into main parts, for instance for different modes of transport. Each leg specifies a transport mode, such as walking or bus, so a route should be partitioned into multiple legs if there are multiple transport modes involved. The main distinction in transport modes is between active and passive modes. Active modes define places where the traveller needs to actively navigate. For our prototypes we have considered just walking, but cycling and driving are other active modes. Passive modes refer to

[3] http://www.opentripplanner.org/
[4] https://developers.google.com/maps/documentation/directions/

places where the traveller is a passenger (i.e. the public transport modes). The ways to provide navigation and actions for deviation detection are different between active and passive modes.

A leg can further be partitioned into *steps*. Active mode legs need step information for giving step-by-step instructions. The step is the unit of instruction, specifying a location for where to give the instruction. Therefore, a step is typically located where there is a change of direction, a need to cross the street, etc. Given the limitations in accuracy of satellite positioning, there should be a lower limit on how close together steps can be placed. Steps placed just a few meters apart will not work well in navigation, as the satellite positioning is likely to place the user at the wrong step. We recommend a minimum distance of 50 meters. For transit legs, steps can be used to define the intermediary stops on the transit route (i.e. telling the user how many stops are left before the destination), and/or give a notification before reaching the destination. The instruction can include a text string, an image for the step (i.e. show a photo of the road to the user) and sound (for voice recording).

While the step locations make up a route to follow, the route data may also specify a *path*. The path is a list of geographical coordinates, typically based on map data and more detailed than the start/end coordinates of steps and legs. Google Directions and OpenTripPlanner include a path in their output for drawing the route on a map, and this will be as detailed as the underlying map data, typically following every bend in the road. As we do not require use of automated route planners, our model does not require a path, but it is included in the conceptual model, and path segments can be provided on either the leg or step level.

Our route data model is meant to be quite flexible, so that routes can be more or less well-defined, they can have time schedules or not, etc. A route defined by a carer will typically have more instruction properties but less formal data such as timing and statistics, and we want applications to be able to use both. Navigation algorithms should make a best effort to guide the user with what data is available.

4.2 Data file specification

For storing and exchanging routes between applications, we define a JSON format with required and optional fields. The main objects are:

```
{
      meta:{…},
      routes:[
            legs:[
                  steps:[…]
            ]
      ]
}
```

The fields of each object are described here. See Appendix 2 for an example JSON file.

Media for instructions (image and sound) can be specified for each step. These are specified with URIs pointing at the resources (image and sound files). Resources stored on the POSEIDON file server are identified by a resource ID. For resources stored on this server, a resource ID can be given instead of a full URI, so that the file server address can be changed without affecting the route.

4.2.1 Relation to existing formats

This format is based on that used by output from Google Directions (GD). It has the same structure of routes containing legs containing steps. Our specification for these elements is a subset of that used by GD. In addition we have custom fields for multimedia instructions.

A summary of the main differences from GD data:

- We have a *meta* object in addition to routes, corresponding to the *trip* level of our conceptual model.
- GD data may have two levels of steps (a list of steps inside a step), while our specification is for one level only (use legs for a high-level partitioning).
- We add step properties for text, image and sound instructions.

Our conceptual model is closer to the output of OpenTripPlanner than it is to GD. Our distinction between leg and step is the same as in OTP, and OTP specifies transport mode on legs and only has one level of steps. An application parsing routes in our and/or the GD format can therefore also be adapted to parse OpenTripPlanner routes with little effort.

4.2.2 meta object

The meta object corresponds to the trip level of the conceptual model, and gives information about the route without being part of the route. The required properties have been selected so that applications only need to parse the meta object, not the route itself, to be able to present a short summary of the route.

JSON identifier	Datatype	Required	Comments
`title`	`string`	Yes	Name for presenting the route in user interfaces.
`start_location`	`string`	Yes	Name of starting point, to tell the user where they need to start from.
`end_location`	`string`	Yes	Name of destination, to tell the user where the route goes.
`start_longitude`	`double`	Yes	Coordinate of starting point.
`start_latitude`	`double`	Yes	Coordinate of starting point.
`end_longitude`	`double`	Yes	Coordinate of destination.
`end_latitude`	`double`	Yes	Coordinate of destination.
`resource`	`string`	No	URI or resource ID for image representing the route.

4.2.3 route object

The JSON file has an array of route objects at the top level, supporting alternative routes as Google Direction does, although we only use single routes in the prototypes. The POSEIDON format only requires an array of leg objects at this level.

4.2.4 leg object

As with the leg level of the conceptual model, each leg object represents a part of the route. Note that, to be compatible with Google Directions, the travel mode is placed in each step. No fields are required, but a meaningful definition of a leg needs either a set of steps or a start location, as a minimum.

JSON identifier	Datatype	Required	Comments
distance	object	No	This object is not required, but if included must have a field *value* with the distance to travel, in meters.
duration	object	No	This object is not required, but if included must have a field *value* with the planned time duration of the leg, in seconds.
start_address	string	No	Street address at start of leg.
end_address	string	No	Street address at end of leg.
start_location	object	No	Coordinates of starting point. This is an object with two properties – lat and lng – each a floating-point coordinate value.
end_location	object	No	Coordinates of destination. Same type of object as start_location.
steps	array	No	List of step objects.

4.2.5 step object

As with the step level of the conceptual model, each step object represents an instruction and the part of the route it applies to. While none of the instruction properties are required, at least one of them should be given for the step to be useful.

JSON identifier	Datatype	Required	Comments
distance	object	No	This object is not required, but if included must have a field *value* with the distance to travel, in meters.
duration	object	No	This object is not required, but if included must have a field *value* with the planned time duration of the step, in seconds.
travel_mode	string	Yes	The prototype system uses the values WALKING and TRANSIT, distinguishing between a mode where the user is actively navigating and a mode where he is a passenger.
start_location	object	Yes	Coordinates of step point – the point the instruction pertains to. This is an object with two properties – lat and lng – each a floating-point coordinate value.
customFilePath	string	No	URI or resource ID for image for the instruction.
customTextInstructions	string	No	Instruction text.
customAudioPath	string	No	URI or resource ID for sound for the instruction.
html_instructions	string	No	Instructions provided by Google Directions, included in the specification for compatibility. Note that it may include html tags.
polyline	object	No	If included, this object must have a property *points,* which is an encoded

			polyline bean for the path (Google Directions provides this).

5 Shopping list data

The POSEIDON framework includes support for aid and training in the usage of money, more specifically buying items and paying the correct amount. For the data storage part, we specify data formats for storing a set of products as well as shopping lists created from these products. These are stored as JSON files on the POSEIDON file server.

5.1 Fields of a shopping list

Field	Datatype	Comments
listName	text	List name
list	array	List of shopping products
assetID	string	ID of SmartPlatform asset (primary user).
cost	number	The total money packed for this shopping list. The amount must be greater than the sum of products price.
List[#].price	number	Price of the product
List[#].name	string	Name of the product
List[#].resourceID	string	resourceID of the image, such that you can retrieve the image with the resourceID and assetID from the file server.
List[#].height	number	Original image file height
List[#].width	number	Original image file width

5.2 JSON example

JSON example for the shopping list data (list.txt):

```
{
      assetID:”85190”,
      ListName:”shopping list one”,
      cost:”25”,
      list:[
            {
                  name:”apple”,
                  price:”12.2”,
                  resourceID:”2f41dc1c-face-4822-b863-ca66a3d3adff”,
                  width:”222”,
                  height:”222”
            },
            {
                  name:”orange”,
                  price:”10.2”,
                  resourceID:”2341dc1c-faee-4822-b863-ca66a3dwefff”,
                  width:”222”,
                  height:”222”
            }
      ]
}
```

5.3 Product information

The product list data has the following fields:

Field	Format	Comments
assetID	text	ID of SmartPlatform asset (primary user).
products	array	List of products
products[#].price	number	Price of the product
products[#].name	string	Product name
products[#].resourceID	string	resourceID of the image, such that you can retrieve the image with the resourceID and assetID from the file server.
products[#].height	number	Original image file height
products[#].width	number	Original image file width

5.4 Usage

In the prototype POSEIDON system, each primary user can have one set of products and one shopping list. The products are stored as a file “products.txt”, in file category “products”. The shopping list is stored as a file “list.txt”, in file category “list”. The system can be extended to allow multiple lists in the same category. The product list is only used by applications which create the shopping list (this is done in the web application for carers in the prototype system). The shopping list is also used by applications which provide training or aid, and contains all the information necessary to provide this functionality.

Shopping lists (files in category “list”) are retrieved with the following URL in the prototype system:

https://ri.smarttracker.no/files/resource.php?type=list

BODY


```
[
    {
        resourceID: "027a0dfe-ce75-48a6-a0a1-9a40f96fb496",
        name: "list.txt",
        mime: "txt",
        md5: "96bb154c772b8b8a7b313c0b4ada0321"
    }
]
```


hide

Updating the list should be done by posting a new list to the same resourceID, to avoid getting multiple list files on the server.

6 Video list data

The framework specifies that secondary users and other carers can define a list of instructional videos. Primary user applications can have video playback available, letting the user select from this list. The videos themselves can be stored either on the file server or on an external service such as YouTube. The play list is defined by a JSON file, in the following format:

```
{
      assetID:<ID of SmartPlatform asset>,
      videos:[
            {
                  type:<"fileServer" or "youtube">,
                  resourceID:<resourceID if type fileServer>,
                  url:<URL for YouTube>
            }
      ]
}
```

Each element in the list is a video. Two types of videos are supported so far. Type "fileServer" means the video is stored as a file on the POSEIDON file server, and a resourceID is given. Such videos should be downloaded to the client device to play. Type "youtube" means the video is available for streaming from the YouTube service.

The play list definition is stored on the POSEIDON file server, in the category "videoList". So it is retrieved with the following URL in the prototype system:

https://ri.smarttracker.no/files/resource.php?type=videoList

7 Calendar data

This chapter has the data specification and other developer information for calendar data in the POSEIDON framework. The architecture and conceptual model is described in D5.1 Development framework, chapter 5.3, as part of the technology infrastructure description. A summary of the architecture is given in the next section.

7.1 Architecture

Figure 10 illustrates the architecture. It shows that calendar events with instructions are defined in the Google Calendar system, while media referenced from event definitions are stored on the POSEIDON file server. An application, for instance a web application, gets events through the Google Calendar HTTP API[5]. It will then get media files referenced in the events from the POSEIDON file server, documented in chapter 3 of this document. When creating a new event, an application will first upload any media files to the file server, enter the returned resource IDs into the event definition, and post that to Google Calendar.

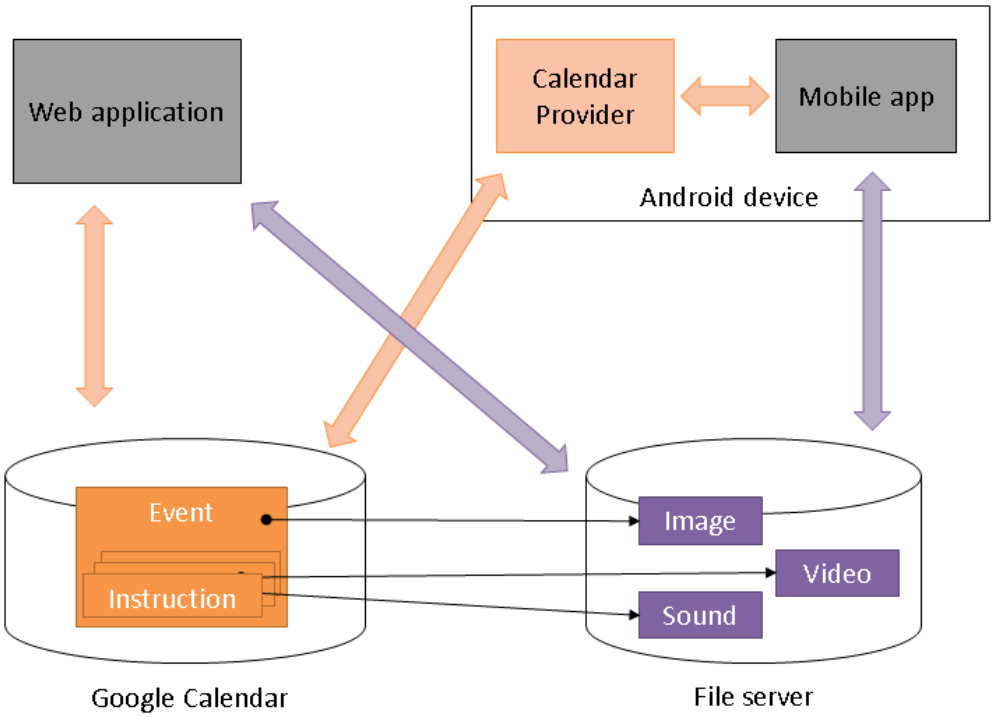

Figure 10: POSEIDON infrastructure for calendar events with instructional media.

On an Android device, applications can use the Calendar Data Provider API[6] of the Android system to interact with the calendar storage. This Android middleware provides an API which is not tied to a specific calendar service, and its implementation handles synchronization of calendar data between cloud stores and the device, simplifying the job of the application programmer. Media files are accessed through the file server API as for other applications.

Here we describe the relevant parts of the Google Calendar and Android Calendar Provider APIs, and the POSEIDON-specific data added. See the referenced online API documentation for specifications and examples of the APIs.

[5] https://developers.google.com/google-apps/calendar/
[6] http://developer.android.com/guide/topics/providers/calendar-provider.html

7.2 Data model

A consequence of the POSEIDON architecture is that the framework supports two different programming interfaces for accessing calendar data. The general principles of the underlying data models of the Google Calendar and Android Calendar Provider are roughly the same, and also similar to that of iCalendar[7]. The main entity is the event. The calendar data models are more complex than it may seem at first glance, as an event can be recurring, and there can be exceptions to the recurrence. POSEIDON uses the basic features of events with reminders. This document goes through the relevant entities, as they appear in the data of the two APIs, and how we use them.

Through the Google Calendar API, data is exchanged in a JSON structure, for instance with an event element containing reminder elements. The Android Data Provider API is basically a database interface, making queries and updates to tables. Figure 11 shows the database schema exposed through this API. It serves as an illustration for the following presentation of data objects.

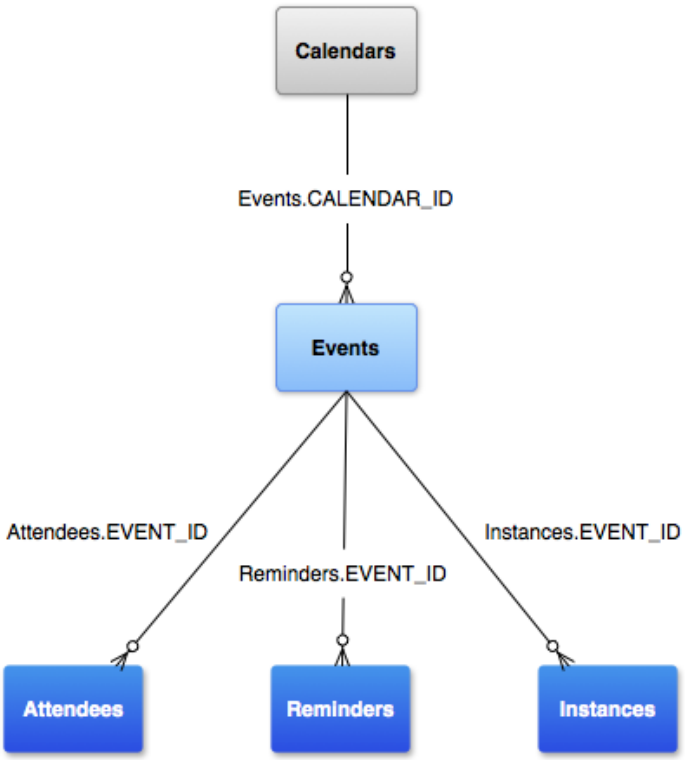

Figure 11: Android Calendar Provider data model.

7.3 Data objects

These are the data objects available through the APIs. We list them with JSON property names for the Google Calendar API and Java constants for the Android calendar provider API.

7.3.1 Calendar

A calendar in this data model is an entity which events belong to – all events belongs to a calendar. A Google account can have any number of calendars. It can also be linked with calendars of other Google accounts. In the Android Calendar Provider API, a device may have multiple accounts with calendar data, and each account may have multiple calendars. Accounts may be of different types,

[7] iCalendar is the closest thing to a standard for exchange of calendar data, see http://en.wikipedia.org/wiki/ICalendar

one of which is Google. So the situation can get quite complex. For the prototype system, each user is restricted to a Google account with a single calendar. The Android device is set up with this Google account, and no other calendar accounts. This is to ensure there is no ambiguity as to where the data is stored, and that the same data is available on the web and on the phone.

7.3.2 Event

The event is the key entity in both the Google and Android calendar data models. The complicating factor is that an event may either be for a single occurrence on a specific date, or it may be recurring, which means that it represents a potentially open-ended amount of occurrences based on some pattern. Being able to represent recurring events is very useful, as it would be very tedious to have to enter the same event every week manually. Therefore we support recurring events in POSEIDON.

An event may also be an "all day" event. This means it doesn't have specific start and end times, but rather cover one or more whole days. In order to keep the primary user interfaces simple, listing events sorted on time, we do not support "all day" events in POSEIDON. For the same reason, we don't want events spanning multiple dates.

The Android calendar API has an instance entity in addition to event. We can query for instances in a given timeframe, and this will retrieve one instance for each occurrence of an event in this timeframe. So the API handles creating instances with specific start and end times for the repeating events. However, it is not possible to insert, update or delete instances, as they are derived from events.

The following table lists the relevant fields for event data, with corresponding Google Calendar JSON names and constants of the Android calendar API. For the Android API, we read data as instances, but insert, update and delete as events.

Google Calendar API	Android Calendar API	Description
	Events.CALENDAR_ID Instances.CALENDAR_ID	Identifier for the calendar the event belongs to.
id	Event._ID Instances.EVENT_ID	For the Android API, a database ID is used to refer to a specific event (needed to update or delete an event).
summary	Events.TITLE Instances.TITLE	Name of the event.
start.dateTime	Events.DTSTART Instances.BEGIN	Datetime for the start of an event/instance.
end.dateTime	Events.DTEND Instances.END	Datetime for the end of an event/instance. A recurring event does not specify this in the Android API.
start.date **end.date**	Events.ALL_DAY Instances.ALL_DAY	An "all day" event specifies start/end as date without time in the Google data. In the Android model there is an allday flag. We do not currently support this in the POSEIDON system.
	Events.DURATION	A recurring event specifies duration for each instance, rather than an end time, in the Android API.
description	Events.DESCRIPTION Instances. DESCRIPTION	This field is meant for a text description of the event. As the POSEIDON framework require a number of

		additional fields, we instead use the description of the Google and Android data as the place to hold POSEIDON-specific data. This is a data structure in JSON format, described in section 7.4.
recurrence[]	Events.RDATE	A recurring event can specify recurring events with this field.
	Events.RRULE	A recurring event can specify rules for recurrence with this field.
location	Instances.EVENT_LOCATION Events.EVENT_LOCATION	Location of the event. The POSEIDON system uses this to connect an event to a route or destination for travel. Details given below.
start.timeZone **end.timeZone**	Events.EVENT_TIMEZONE	A time zone is specified when creating an event in the Android API. Google data has separate timeZone values for start and end times.

An event can be connected to a location or planned route stored on the POSEIDON file server. Each location configured for the user has a name. If navigation is added to an event, the location or route name is stored in the location field of the event. An event with location is interpreted as the planned time to start travelling to this location. A calendar application capable of starting navigation functionality should look for the location in the list of configured locations or routes of the user, and invoke navigation if a route is found.

7.3.3 Reminder

An event can have one or more reminder objects. This object specifies that the user should be notified of an upcoming event a specific number of minutes before event start. Calendar applications for the primary user must notify the user at the specified time.

The reminder entity is very simple, without any own message. Google Calendar can include default reminders which are applied to all events, unless the event data explicitly specifies not to use defaults. The default reminders are configured in Google Calendar, independently of POSEIDON. If used, such reminders will show up in the Android data.

In the Google Calendar API, reminders are included in the data structure for an event. Any number of event-specific reminders can be added to an array in this data. In the Android API, the reminder is a separate entity. Each reminder entity has a reference to the event it belongs to.

Google Calendar API	Android Calendar API	Description
	Reminders.EVENT_ID	For the Android API, a reminder entity is retrieved with a reference to the event it belongs to.
useDefault		Google Calendar has a configuration for default reminders, applied to all events unless this flag is false.
overrides[].method	Reminders.METHOD	Google Calendar supports several reminder methods. Email and SMS methods are supported in both APIs, and handled by Google. We do not recommend using these reminder types

		for our user group. The type “popup” in the Google data, corresponding to Reminders.METHOD_ALERT in Android, is the type to handle with notifications in an application.
overrides[].minutes	Reminders.MINUTES	Number of minutes before event start.

7.3.4 Attendee

For completeness we also include the attendee object. It represents a person or other entity which should be present at an event, mainly identified via email address. The POSEIDON framework does not make any special use of this object. It is useful for adding a social aspect to the calendar, specifying people to meet.

7.4 POSEIDON data extension

The POSEIDON framework adds additional data to events, in the form of event media and a list of instructions. To fit this into the data model of the Google and Android calendars, we put this data in the event’s description field. It is stored as a JSON structure, with the following possible content:

```
{
      description:<event description>,
      eventIcon:<resourceID or URL>,
      eventVideo:<resourceID or URL>,
      instructions:[
            {
                  index:<place in sequence>,
                  title:<instruction text>,
                  image:<resourceID or URL>,
                  audio:<resourceID or URL>,
                  video:<resourceID or URL>
            }
      ]
}
```

An event can have a list of instructions, where each should have at least one of title, image, audio or video. These are shown to the user in sequence. All media can be specified either as a URL to any online resource, or just as a resource ID for a file on the POSEIDON file server. Each element is also described in the table below.

JSON identifier	Datatype	Description
description	string	A text description for the event.
eventIcon	string	An image to represent the event, specified by URL or resource ID.
eventVideo	string	A video, as an audio-visual alternative to the description (URL or resource ID).
index	integer	Each instruction can explicitly specify its place in the sequence of instructions (low to high number), otherwise the sequence is given by the order in the JSON array.
title	string	Text of the instruction.
image	string	Image of the instruction; URL or resource ID.
audio	string	Audio of the instruction; URL or resource ID.
video	string	Video of the instruction; URL or resource ID.

Appendix 1: SmartPlatform API resources

We include here a documentation of the most important resources (data objects) of the data model and REST API of Tellu SmartPlatform. The resource sections are ordered alphabetically, and list all fields available through the API. The complete REST API documentation is available online, and should be consulted for application development, as it gives a complete and up-to-date listing. It is available at the following URL:

https://telludoc.atlassian.net/wiki/display/SMARTTRACKER/Smarttracker+API+v3

Alarm

An alarm is a notification that requires the attention of a user, usually generated by the reasoning engine based on some rule.

Property	Type	Optional	More info	Filtering
name	string	-		-
owner	customer	-		-
dateCreated	date	-		equals, greater, less
lastUpdated	date	-		
comment	string	-		equals, contains
ackNeeded	boolean	-		equals
logLevel	integer	-	Degree of severity of alarm. 0 is most severe, -20 least. -20 should be without immediate notification.	equals, greater, less
asset	asset	-	Asset associated with the alarm.	equals (id of asset)
ackedBy	user	-		
rule	rule	-		
alarmCenter	customer	-		
trigger	position	-	The observation triggering the rule/alarm, if available.	
position	position	-	The position of the related asset when the alarm was created.	
zone	zone	-	Zone relevant to the triggering of this alarm.	-

Asset

An asset represents a person tracked by the system (the primary user). The *properties* and *tags* fields are the most important to the other modules in POSEIDON, as these are where the user data and state is stored.

Property	Type	Optional	More info	Filtering
name	string	NO		equals, contains
description	string	YES		equals, contains
owner	customer	-		-
lastValidPosition	position	-	The most recent observation with a valid position received by the position provider of the asset.	-
lastPosition	position	-	The most recent observation received by the position provider of the asset. If the position is valid this will be the same as lastValidPosition.	-
tags	list of objects	YES	When creating or updating only the name of the tag will be used, the time will be set by the server.	contains
groups	list ofgroup	YES	When creating or updating only the id of the group will be used.	-
insideZones	list ofzone	-		contains (id of zone)
icon	string	-	This icon is the icon set by the asset's type.	-
image	string	-		-
type	type	YES	When creating or updating only the id of the type will be used.	equals (id of type)
tracked	boolean	YES	If enabled all observations received by the position provider will be stored. Cannot be set to true if the trackMode is "never", cannot be set to false if the trackMode is "always".	-
trackMode	string	YES	"always" will always store observations received. Rules	-

Property	Type	Optional	More info	Filtering
			cannot change whether or not the asset is tracked. "never" will never store observations received permanently. Rules cannot change whether or not the asset is tracked. "manual" depend on the tracked property to determine if observations are stored. Rules can change whether or not the asset is tracked.	
properties	list of objects	YES	The possible property names are based on the configured properties in the asset's type.	contains
alarms	list ofalarm	-	The five most recent, unacknowledged alarms. Useful for creating lightweight clients.	-
deviceCommands	list of objects	-	TODO	-
positionProvider	tag ordevice	YES	When creating or updating only the id of the object will be used. The server will first attempt to find a device matching the id and if not found a tag.	equals (id of positionProvider)

Device

A device specifies a source of sensor data, and is assigned to an asset to provide sensor data for that asset. In POSEIDON the app running on the user's tablet will be one such "device", and any other client sub-system posting data through an edge will also have a device entry.

Property	Type	Optional	More info	Filtering
name	string	NO		
description	string	YES		
owner	customer	-		
lastValidPosition	position	-		
lastPosition	position	-		
sensorDeviceType	string	NO		

Property	Type	Optional	More info	Filtering
active	boolean	-		
uuid	string	-		
primaryProperties	object	NO		
commandProperties	object	-		
additionalProperties	object	YES		
filters	list of filter	YES		

Group

Assets can belong to groups, which may be useful for group logic (trigger rules for all assets in a group).

Property	Type	Optional	More info	Filtering
name	string	NO	Any non-empty string. Cannot be the same as any existing group inside the customer.	equals, contains
description	string	YES		equals, contains
owner	customer	-		
assets	list of assets	YES		

Position

Although called position, this is more generally a sensor observation. The *properties* field can contain whatever key-value pairs the source wants to post.

Property	Type	Optional	More info	Filtering
asset	asset	-		equals
valid	boolean	-		equals
latitude	double	-		equals, greater, less
longitude	double	-		equals, greater, less

Property	Type	Optional	More info	Filtering
accuracy	integer	-	Estimated accuracy in meters	equals, greater, less
speed	integer	-	Speed as reported by device in meter per second.	equals, greater, less
address	string			-
timestamp	date			equals, greater, less
properties	object			-
events	list of string			-
insideZones	list ofzone	-	A list of zones the position was inside when it was received by the system.	contains (id of zone)

Rule

A rule is a configurable unit of logic for the reasoning engine. The active set of rules defines the service behaviour. The data available through the API is mainly for viewing and changing rule states (turn on and off).

Property	Type	Optional	More info	Filtering
name	string	-		equals, contains
description	string	-		equals, contains
owner	customer	-		-
status	string	YES	Values: "active", "inactive", "unknown", "stopped", "failed", "requiresConfiguration". When changing the status, only "active" or "inactive" are valid inputs.	-

Type

Assets can be typed, with the type specifying what properties an asset has. POSEIDON assets will have their own type, specifying the fields needed for our system.

Property	Type	Optional	More info	Filtering
name	string	NO	Any non-empty string. Cannot be the same as any existing type inside the customer.	equals, contains
description	string	YES	Any string.	
owner	customer	-		
icon	string	YES	URL referring to an icon describing assets of this type.	
properties	object	YES	Object where each key is a different property name. Property value is an object with at least two entries, type and valueType. If the type is a list it must also include a list of string called items.	

Zone

Zones are used to define location-specific logic such as geofence (trigger a rule on entering or leaving an area).

Property	Type	Optional	More info	Filtering
name	string	NO	Any non-empty string. Cannot be the same as any existing zone inside the customer.	equals, contains
description	string	YES		equals, contains
owner	customer	-		
position	latlon	-	An object with two entries, latitude and longitude.	
singleLevel	boolean	YES	When checking if an asset is inside, do they need to be on the same floor.	
floor	integer	YES		
textual	string	YES		
address	string	YES		
points	list of latlon	NO	List of objects with two entries, latitude and longitude. Must have at least 3 objects.	

Property	Type	Optional	More info	Filtering
assets	list ofassets	-	A list of assets that have received an observation with a valid position after this zone was created.	

Appendix 2: Route data files

This appendix lists a formal definition and an example of route data.

Route file schema for POSEIDON mobile application

The prototype mobile application uses a formal definition to do validation and parsing of route files. This definition is in a proprietary XML format, which is not JSON-specific, but it is included here as it is a formal definition in a straightforward form.

```
<group name="root">
  <group name="meta" cardinality="1">
    <item name="title" type="string" cardinality="1"/>
    <item name="start_location" type="string" cardinality="1"/>
    <item name="end_location" type="string" cardinality="1"/>
    <item name="start_longitude" type="double" cardinality="1"/>
    <item name="start_latitude" type="double" cardinality="1"/>
    <item name="end_longitude" type="double" cardinality="1"/>
    <item name="end_latitude" type="double" cardinality="1"/>
    <item name="resource" type="string" cardinality="0-1"/>
  </group>
  <group name="routes" cardinality="1-n">
    <group name="legs" cardinality="1-n">
      <group name="distance" cardinality="0-1">
        <item name="value" type="integer" cardinality="1"/>
      </group>
      <group name="duration" cardinality="0-1">
        <item name="value" type="integer" cardinality="1"/>
      </group>
      <item name="start_address" type="string" cardinality="0-1"/>
      <item name="end_address" type="string" cardinality="0-1"/>
      <group name="start_location" cardinality="0-1">
        <item name="lat" type="double" cardinality="1"/>
        <item name="lng" type="double" cardinality="1"/>
      </group>
      <group name="end_location" cardinality="0-1">
        <item name="lat" type="double" cardinality="1"/>
        <item name="lng" type="double" cardinality="1"/>
      </group>
      <group name="steps" cardinality="0-n">
        <group name="distance" cardinality="0-1">
          <item name="value" type="integer" cardinality="1"/>
        </group>
        <group name="duration" cardinality="0-1">
          <item name="value" type="integer" cardinality="1"/>
        </group>
        <item name="travel_mode" type="string" cardinality="1"/>
        <group name="start_location" cardinality="1">
          <item name="lat" type="double" cardinality="1"/>
          <item name="lng" type="double" cardinality="1"/>
        </group>
        <item name="customFilePath" type="string" cardinality="0-1"/>
        <item name="customTextInstructions" type="string" cardinality="0-1"/>
        <item name="customAudioPath" type="string" cardinality="0-1"/>
        <item name="html_instructions" type="string" cardinality="0-1"/>
        <group name="polyline" cardinality="0-1">
          <item name="points" type="string" cardinality="1"/>
        </group>
      </group>
    </group>
  </group>
</group>
```

Example route JSON

The following is an example of a JSON route according to POSEIDON specification.

```
{
   "meta":{
            "title":"Test route",
            "start_location":"Karde",
            "end_location":"Problemveien",
            "start_longitude":10.7163861111111,
            "start_latitude":59.9438666666667,
            "end_longitude":10.72225,
            "end_latitude":59.9430305555556,
            "resource":"9fbe5760-90be-4f4e-b886-8d24a35b11f6"
   },
   "routes":[
      {
         "legs":[
            {
               "distance":{
                  "text":"0,5 km",
                  "value":"488"
               },
               "duration":{
                  "text":"6 min",
                  "value":"324"
               },
               "end_address":"Problemveien",
               "end_location":{
                  "lat":"59.9430305555556",
                  "lng":"10.72225"
               },
               "start_address":"Karde",
               "start_location":{
                  "lat":"59.9438666666667",
                  "lng":"10.7163861111111"
               },
               "steps":[
                  {
                     "distance":{
                        "text":"0,5 km",
                        "value":"488"
                     },
                     "duration":{
                        "text":"6 min",
                        "value":"324"
                     },
                     "travel_mode":"WALKING",
                     "start_location":{
                        "lat":"59.9438666666667",
                        "lng":"10.7163861111111"
                     },
                     "end_location":{
                        "lat":"59.9438666666667",
                        "lng":"10.7163861111111"
                     },
                     "customFilePath":"c3f0ec11-c98e-4e1a-9726-c52fa7ca00d9",
                     "customTextInstructions":" ",
                     "customAudioPath":" "
                  },
                  {
                     "distance":{
                        "text":"0,5 km",
                        "value":"488"
                     },
                     "duration":{
                        "text":"6 min",
                        "value":"324"
                     },
                     "travel_mode":"WALKING",
                     "start_location":{
                        "lat":"59.9431138888889",
```

```
            "lng":"10.71595"
         },
         "end_location":{
            "lat":"59.9431138888889",
            "lng":"10.71595"
         },
         "customFilePath":"30804632-005b-4d4b-9042-6f908c2017e8",
         "customTextInstructions":" ",
         "customAudioPath":" ",
         "polyline":{
            "points":"mszlJ{}k`AXT@UMGOIDg@PqB\\yA"
         }
      },
      {
         "distance":{
            "text":"0,5 km",
            "value":"488"
         },
         "duration":{
            "text":"6 min",
            "value":"324"
         },
         "travel_mode":"WALKING",
         "start_location":{
            "lat":"59.9428361111111",
            "lng":"10.717275"
         },
         "end_location":{
            "lat":"59.9428361111111",
            "lng":"10.717275"
         },
         "customFilePath":"69ff0f32-511c-4b05-b9e3-7cb640d86445",
         "customTextInstructions":" ",
         "customAudioPath":" ",
         "polyline":{
            "points":"yqzlJafl`ABOLRFMz@uCp@}BL{@DiB@q@TkAPu@"
         }
      },
      {
         "distance":{
            "text":"0,5 km",
            "value":"488"
         },
         "duration":{
            "text":"6 min",
            "value":"324"
         },
         "travel_mode":"WALKING",
         "start_location":{
            "lat":"59.9418722222222",
            "lng":"10.7204722222222"
         },
         "end_location":{
            "lat":"59.9418722222222",
            "lng":"10.7204722222222"
         },
         "customFilePath":"f03c983f-c186-42d9-b7df-19dbe48fd2a3",
         "customTextInstructions":" ",
         "customAudioPath":" ",
         "polyline":{
            "points":"skzlJyyl`AHm@@OKH[@UKa@w@sBuD_@q@"
         }
      },
      {
         "distance":{
            "text":"0,5 km",
            "value":"488"
         },
```

```
                        "duration":{
                           "text":"6 min",
                           "value":"324"
                        },
                        "travel_mode":"WALKING",
                        "start_location":{
                           "lat":"59.9430305555556",
                           "lng":"10.72225"
                        },
                        "end_location":{
                           "lat":"59.9430305555556",
                           "lng":"10.72225"
                        },
                        "customFilePath":"9fbe5760-90be-4f4e-b886-8d24a35b11f6",
                        "customTextInstructions":" ",
                        "customAudioPath":" ",
                        "polyline":{
                           "points":"{rzlJ_em`A"
                        }
                     }
                  ]
               }
            ]
         }
      ]
}
```