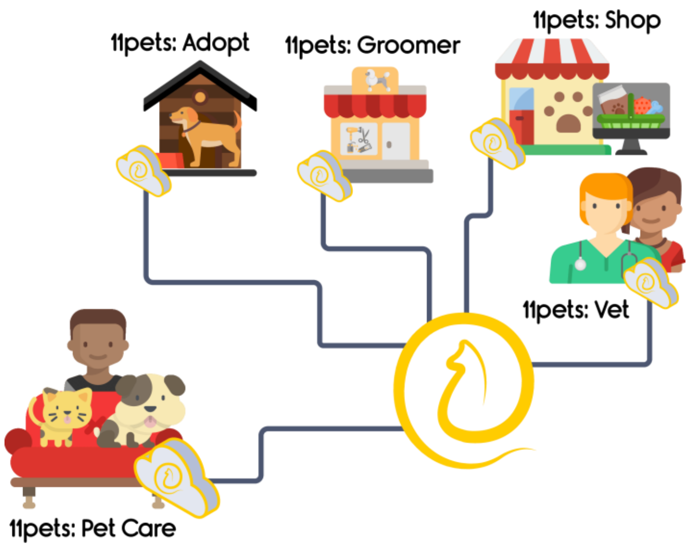
Data Management for Knowledge Extraction

Demos Pavlou (demos.pavlou@11pets.com), Georgios Moullotos (giorgos.moullotos@11pets.com),
Kyriakos Stavrou (kyriakos.stavrou@11pets.com)

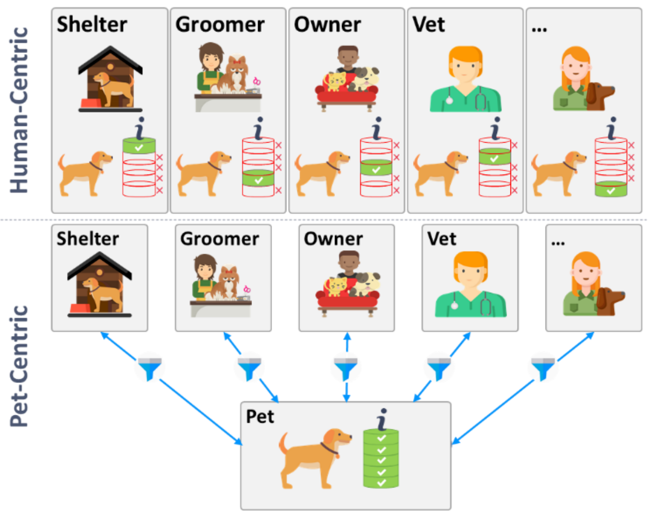
The requirements



11pets is the leading software ecosystem for the pet industry offering solutions for pet families, pet professionals and pet welfare organizations. We offer web-based and mobile application solutions that help the different actors manage the data of their pets and their day-to-day business needs.

Our large userbase and the wide usage of our system, allowed us to collect a significant volume of interdisciplinary data (>10M data points) from different sectors of pet-care including care at home, by veterinarians, groomers, and shelters. By analyzing this pool of data, we have published multiple industry reports that drew the attention of key enterprises. These reports, however, are only based on descriptive analysis of the data, that is, statistical analysis with no knowledge extraction.

Pet-centric Vs Human Centric approach



Today's systems are *human-centric*; they are built around the pet professionals rather than the pets and their families. The current approach is equivalent to a national health system where each doctor, hospital, school, etc. has its own, *isolated* copy of each person's data and is not able to share it with anyone else. This data fragmentation makes it impossible to provide optimal care as each entity has just a subset of the information and nobody can see the whole picture.

In a *pet-centric* system, there is one, consistent copy of the pet's data that includes all information with each entity having a different view. This allows structured and standardized sharing of information. Moving the center of pet-care from the service providers to the pet and its family allows collaboration between each entity. Teamwork means more information for professionals and efficiency in the identification and solution of issues providing the breakthrough the industry needs. Imagine a pet that had an allergic reaction while in the shelter and its new vet is not aware. It is only in a pet-centric system that continuity is guaranteed and professionals have access to all the necessary information.

11pets was the first to introduce the pet-centric approach to the industry and is today the most widely used digital pet-care platform. The 11pets platform enables collaboration at every level considering all privacy constraints. Professionals and families select which data they will share, with whom, and for how long. Each time we release a new product and cover another sector of the industry, we enrich our data not only with additional information but also, with additional parameters.

Knowledge extraction using pet-care data

Machine learning and knowledge/intelligence extraction are widely used today in a variety of areas, including healthcare, security, transportation, risk detection, risk management, etc. [ref]. The field had major contributions in the areas of self-driving cars, natural language translation, and healthcare [ref]. It is a field that is growing rapidly and each day is applied to more real-life problems.

Recently, there have been several efforts on using data analysis for the pet industry [ref, ref]. However, these studies cover only one pet-care sector at a time as their datasets are limited to it. As explained already, modern pet-care involves different sectors that collaboratively care for each pet. The pet-centric nature of the 11pets platform allows all these care providers to work together by centralizing the data management. This, in turn, enables having composite, consistent and coherent interdisciplinary data from different sectors, making it unique. An example of a study that is only enabled by this pet-centric approach is the identification of the relationship between the care a pet receives at the groomer and any skin allergies later treated by a veterinarian.

Role-Based Access Control (RBAC) in multitenant environments where a tenant can have multiple users with different roles is not a sufficient solution for us, since we require to Service Level Agreements (SLA) to control the features that are permitted by a user as well as finer control over actions because we want to allow users to create additional roles on a per tenant basis, thus limiting the functionality they expose to their employees based on their business needs.

The requirements are as follows:

- A user with a single account can have access to multiple projects, e.g., a shop and a groomer
- A user can have different roles in different projects
- A role can perform only a limited set of functions in the system and see only a limited amount of information.
- A project can have only a subset of the functionality of the system, which is defined by the Service Level Agreement that is mandated by the subscription type.

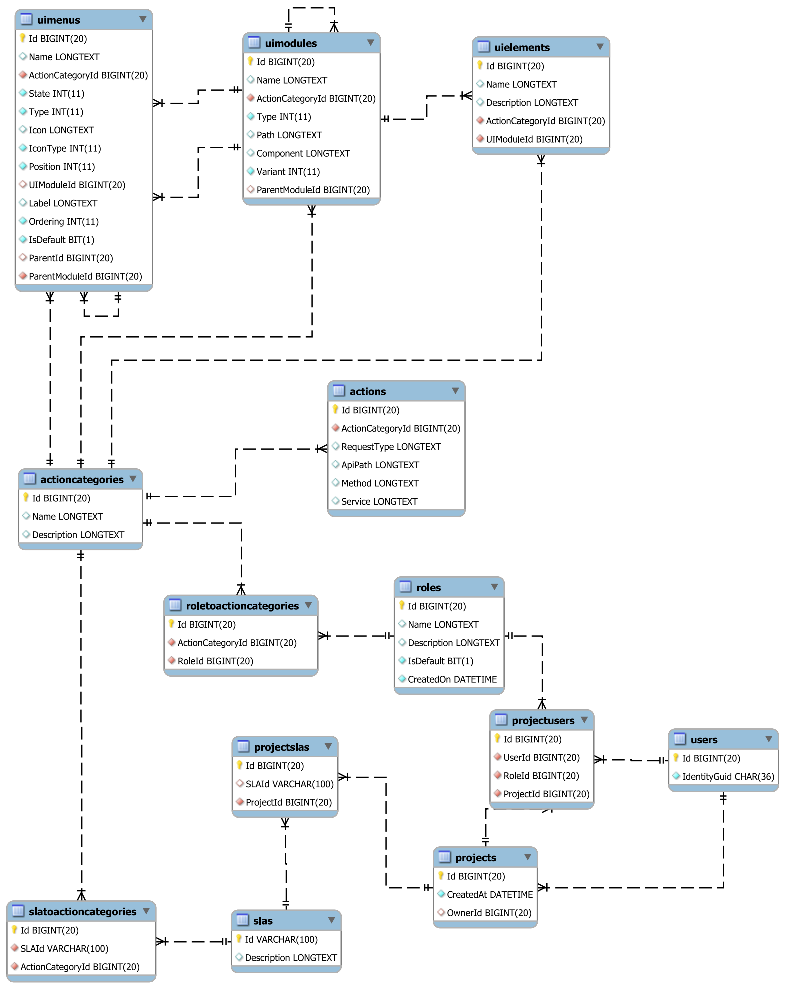
Database design

The Entity Relationship Diagram below shows the most important parts of the database design implemented to satisfy the requirements. A Project (tenant) has multiple Users, and each User can belong to a role for the specific project, allowing users to belong to multiple projects with different roles. A project has multiple SLAs based on the subscriptions of the tenant. Actions are used to give permissions on sub processes of the system. These sub processes could be an API endpoint or a message handler. Actions are grouped into Action Categories to compose features of the platform. A role can have multiple Action Categories assigned. Finally, an Action Category can have UIModules, UIMenus and UIElements assigned that are used by the front-end to construct the dashboard of the user based on the permissions he/she has.

Administration of permissions

The system exposes all available Actions to the administrator. An Action could be the name of the request or an arbitrary string depending on the conventions used in the project.

The administrator can group Actions into ActionCategories, for example the ActionCategory ManageInvoices can include Actions RetrieveInvoices, RefundInvoices, GenerateReport, while the ActionCategory ViewInvoices could include only RetrieveInvoices. Using ActionCategories the administrator can create meaningful groups of permitted actions that meet the business needs for the features offered by the platform.



An ActionCategory can be assigned to Roles. For example a Role Manager can have ManageInvoices while a Role Employee can have access to ViewInvoices, or no access at all.

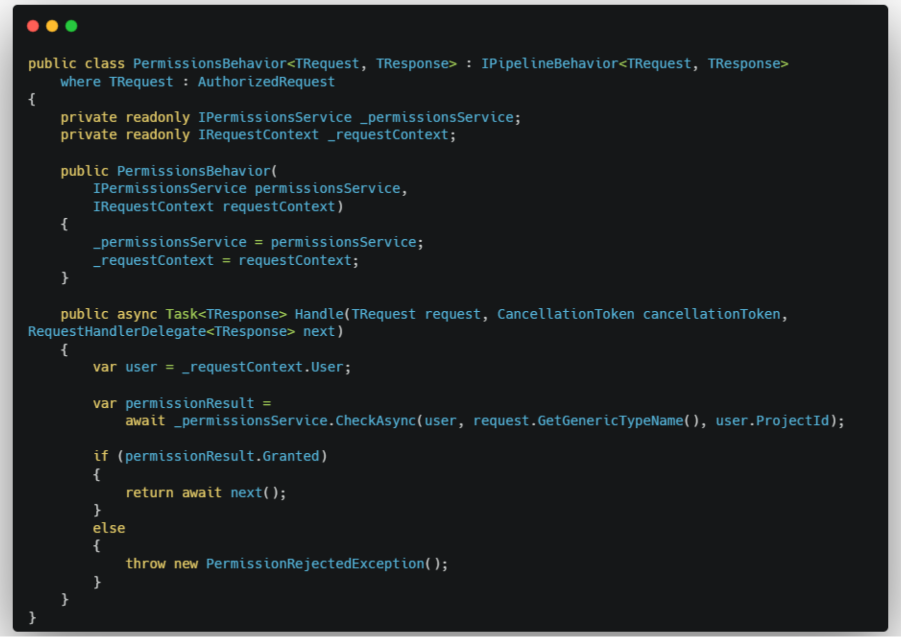
Roles can be used to assign fine-grained permissions to users for a project. When a User is assigned to a project a Role must be provided.

A SaaS platform must have the ability to limit access to features based on the SLA of the subscription of a tenant. Each SLA has a set of permitted actions assigned to it.

Consider a platform that offers a Basic and a Premium Plan. The Premium Plan allows the GenerateReport Action while the Basic plan does not. We need to ensure that the Role Manager cannot GenerateReport even though it is permitted by the Role. This can be achieved by calculating the intersection of the set of permitted actions defined in the Role and the set of permitted actions assigned by the SLA.

Permission checks

Permissions are usually a cross-cutting concern for an application. In our case we use pipelines of filters to enforce permissions. In projects where Mediator pattern is used, we register a pipeline step that for every message we check if the permission exists, otherwise we return an error. In projects using ASP.Net MVC we could define an action filter that checks based on the route or method name.

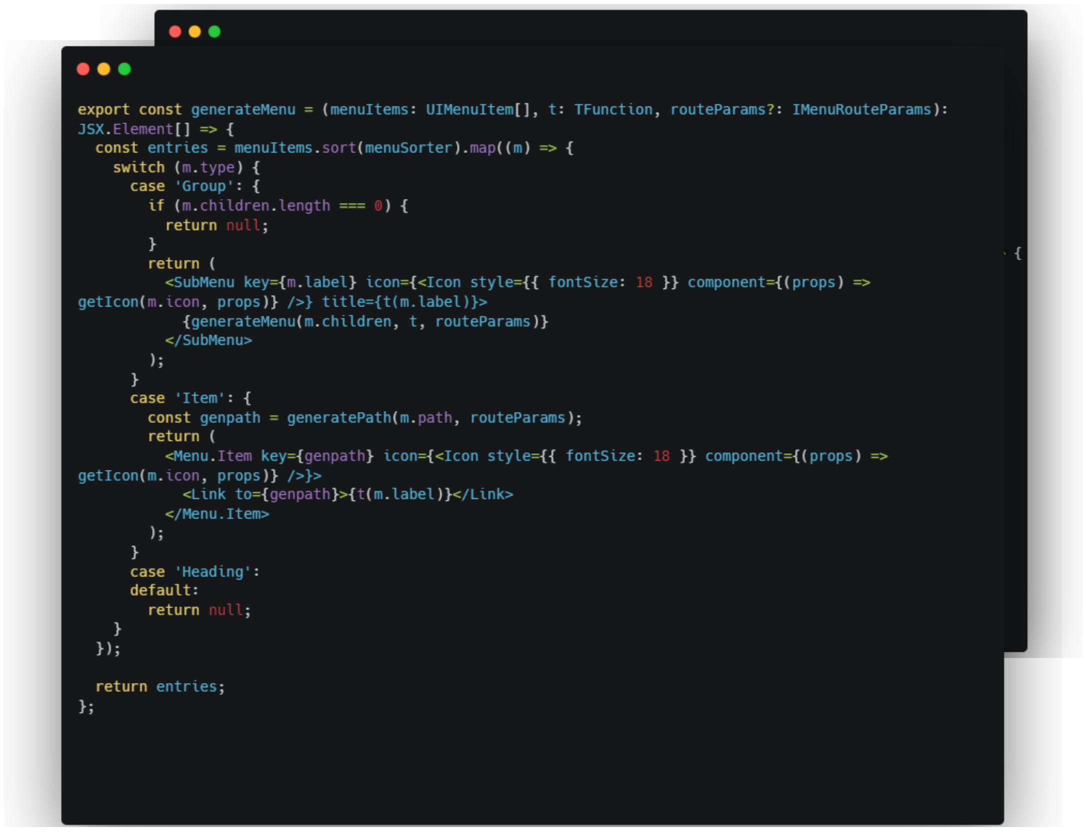


```
public class PermissionsBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
    where TRequest : AuthorizedRequest
{
    private readonly IPermissionsService _permissionsService;
    private readonly IRequestContext _requestContext;

    public PermissionsBehavior(
        IPermissionsService permissionsService,
        IRequestContext requestContext)
    {
        _permissionsService = permissionsService;
        _requestContext = requestContext;
    }

    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken,
RequestHandlerDelegate<TResponse> next)
    {
        var user = _requestContext.User;

        var permissionResult =
            await _permissionsService.CheckAsync(user, request.GetGenericTypeName(), user.ProjectId);

        if (permissionResult.Granted)
        {
            return await next();
        }
        else
        {
            throw new PermissionRejectedException();
        }
    }
}
```

Blocking actions in the backend is sufficient to guarantee that users cannot perform undesired actions. To provide a good User Experience (UX) we need to either hide elements on the UI or be able to disable them. When building a Single Page Application (SPA) the client-side code needs to be aware of the restrictions imposed by permissions to adapt to what each user can do.

Pets.Platform.Permissions provides a solution that allows to calculate UI Menus, Routes and accessible UI Elements based on the access level of the user. The backend should provide the client-side code with a set of Routes, Menus and Elements that can be used to construct the correct dashboard. It is the responsibility of the front-end to provide a good UX.

A UIModule has a coarse scope and is mainly used to guard pages. The UIModule is assigned an ActionCategory and can only be accessed if the user is permitted based on Role and SLA. The Front-end can then display the normal page or a forbidden message if the user accesses the route. In our dashboards using react-router we use the following guard to avoid registering them.



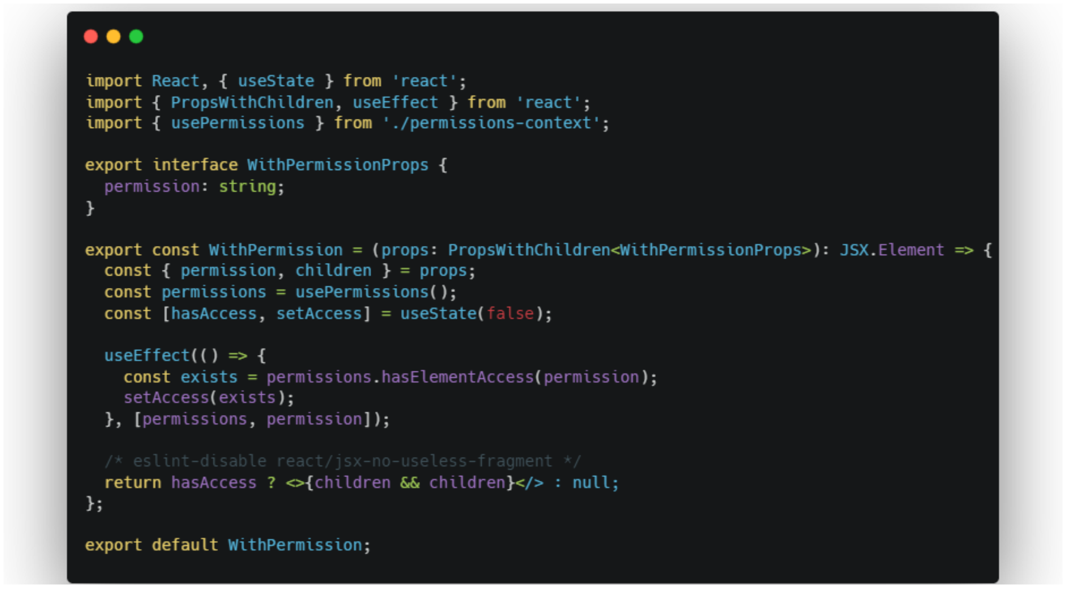
```
export const generateMenu = (menuItems: UIMenuItem[], t: TFunction, routeParams?: IMenuRouteParams):
JSX.Element[] => {
  const entries = menuItems.sort(menuSorter).map((m) => {
    switch (m.type) {
      case 'Group': {
        if (m.children.length === 0) {
          return null;
        }
        return (
          <SubMenu key={m.label} icon={<Icon style={{ fontSize: 18 }} component={(props) =>
getIcon(m.icon, props)} />} title={t(m.label)}>
            {generateMenu(m.children, t, routeParams)}
          </SubMenu>
        );
      }
      case 'Item': {
        const genpath = generatePath(m.path, routeParams);
        return (
          <Menu.Item key={genpath} icon={<Icon style={{ fontSize: 18 }} component={(props) =>
getIcon(m.icon, props)} />}>
            <Link to={genpath}>{t(m.label)}</Link>
          </Menu.Item>
        );
      }
      case 'Heading':
      default:
        return null;
    }
  });

  return entries;
};
```

UIMenus can control the sidebar or navbar menus of the dashboard. The backend provides a tree of MenuItems which can be used to build the correct menus for the user.

Finally, UI Elements can be used for find-grained control for buttons, or sections in the page. For example, the GenerateReport button can be disabled or hidden if the user does not have the appropriate permissions. A guard like the following could be used to ensure that an element is hidden.

Performance

Performance is always a concern. With a traffic of thousands of users, we need to reduce the number of round trips to the database for permission checks. Permissions do not change very often which makes them a good candidate for caching. We implemented a provider class that reads the required records from the database and caches them in memory to perform the checks faster.



```
import React, { useState } from 'react';
import { PropsWithChildren, useEffect } from 'react';
import { usePermissions } from './permissions-context';

export interface WithPermissionProps {
  permission: string;
}

export const WithPermission = (props: PropsWithChildren<WithPermissionProps>): JSX.Element => {
  const { permission, children } = props;
  const permissions = usePermissions();
  const [hasAccess, setAccess] = useState(false);

  useEffect(() => {
    const exists = permissions.hasElementAccess(permission);
    setAccess(exists);
  }, [permissions, permission]);

  /* eslint-disable react/jsx-no-useless-fragment */
  return hasAccess ? <>{children}</> : null;
};

export default WithPermission;
```

Other concerns

Permissions can change over time, therefore the administrator should have the ability to reset and reload the permissions while the system is running without having to restart all the services. To achieve this, a message to reload permissions should be consumed by the service.

Other concerns for a platform are subscription changes of a tenant and user assignments to roles or tenants. The microservice managing this should publish appropriate events that the permissions module should consume to update the database and its caches.

Code

The source code of our solution can be found here

Acknowledgements

"This work was co-funded by the European Union and the Republic of Cyprus through the Research and Innovation Foundation (Project: INNOVATE-COVID/0420/0018)".