Project documentation - SPrice

Motivation

Vietnamese community's most common livelihood is grocery retailing. Their business activities have a significant impact on the Czech grocery market. As of today, their workflow requires manually navigating in the remarkably large set of products and offers.

Solution

The SPrice mobile app is trying to help businesses by automating the task of searching and comparing products offered in local supermarkets.

Feature set

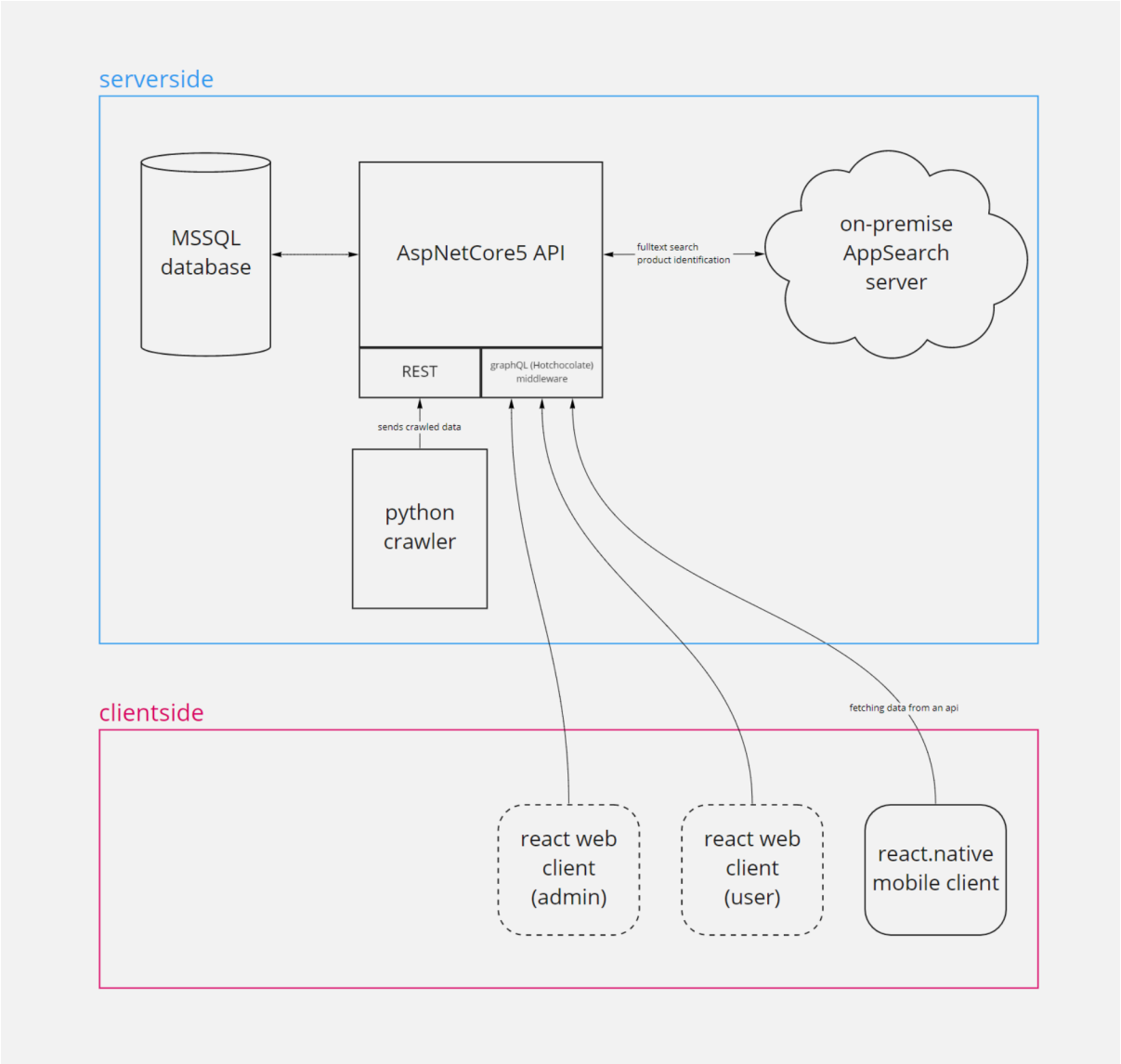
- Search for a product and its offerings
- Filter products based on category or where its sold
- Create shopping lists with offers
- Adding product to a favourite list

What to expect in future updates?

- Shopping list templating
    - Generating a shopping list from a set of specified products
    - Generated list will be chosen based on price, location and other criteria
- Recommendation
    - Homescreen composed of relevant offers chosen by a recommender component
- Notifications
    - New offer of a product labeled as favourite
    - Offers that shouldn’t be missed chosen by recommender component
- 3rd party login
    - Login via facebook, apple, google
- Improved offer filtering
    - Allowing combination of filters
- Placing in-app orders
- Price development visualisation

Architecture

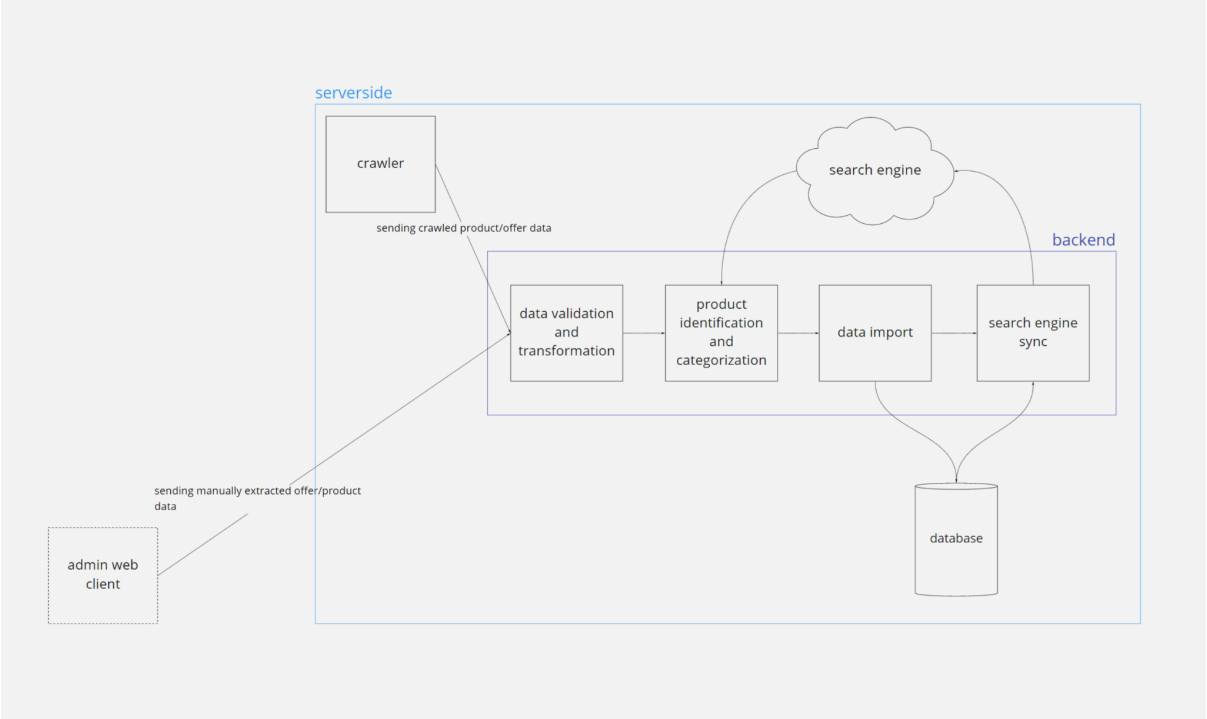
The solution is based on client-server architecture with few additional components.



Primary processes

Data import

The data - specifically product information - are not uniformed nor structured and because of that it can be non-trivial to parse the product data and identify the entities.
There are 2 main sources of offer and product data - **vendor's websites** and **paper leaflets**. Websites are automatically processed by a **crawler** program, which after extraction sends data to the **backend server**. **Leaflets** are processed **manually** and sent to the server using an **admin web client**.
Server then has to **validate and transform** incoming data. Each offer (and its corresponding product) needs to be **identified** (using an EAN code or product identity mechanism) and **categorized**.
After that, offers and new products (that weren't in the database prior to the corresponding import) are **added to the database**. New products also have to be **added to search engine's storage** to be searchable by the user.

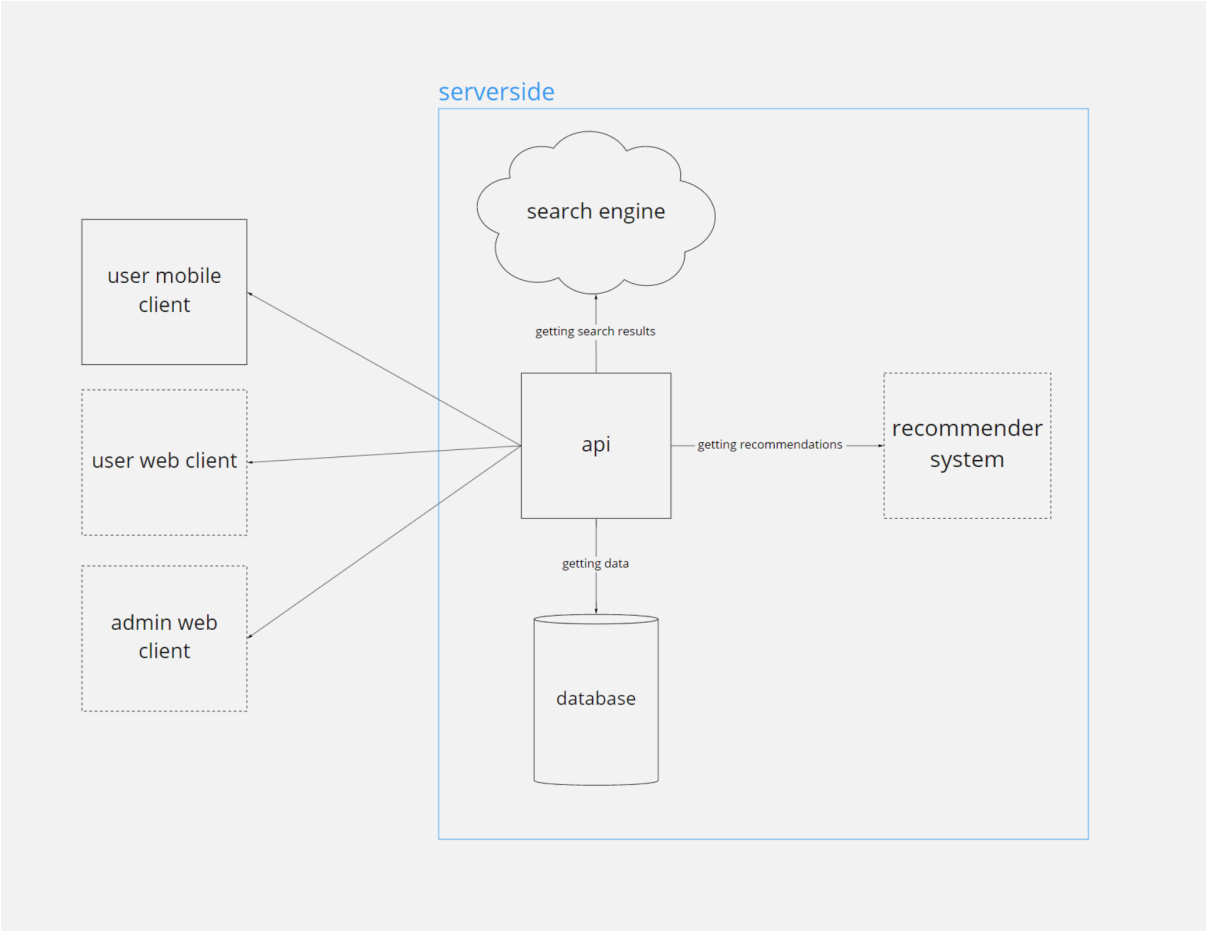


Product identification and categorization

Search engine's heuristics (product name similarity)can be used to get **matched products**. **Certain threshold** is defined as a confidence level for a complete match considered as identification, or product similarity.
From the result of the matching mechanism **matching products and categories** can be determined.
From a vendor's catalog we use an already **existing categorization**. We will need to define a **category mapping** from vendor's categories to internal categories.

Client application data flow

Data displayed by the client is **requested** from the **backend application**. Backend then **processes incoming requests** and executes corresponding **business logic** and sends back the result. During that procedure, the backend application usually has to **access data** from an **external component** (search engine, database or recommender component).

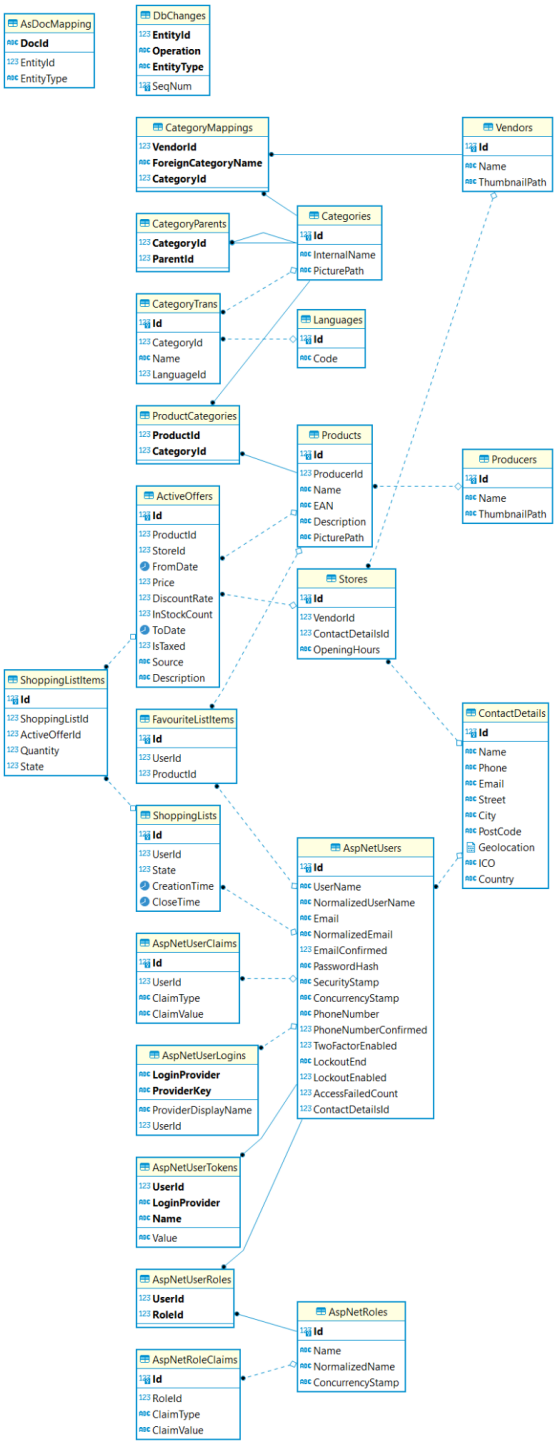


Note that dashed rectangles represent components that will be implemented in the future

Data modelling

Data primarily stored in a relational database.

Data model



Components

Backend

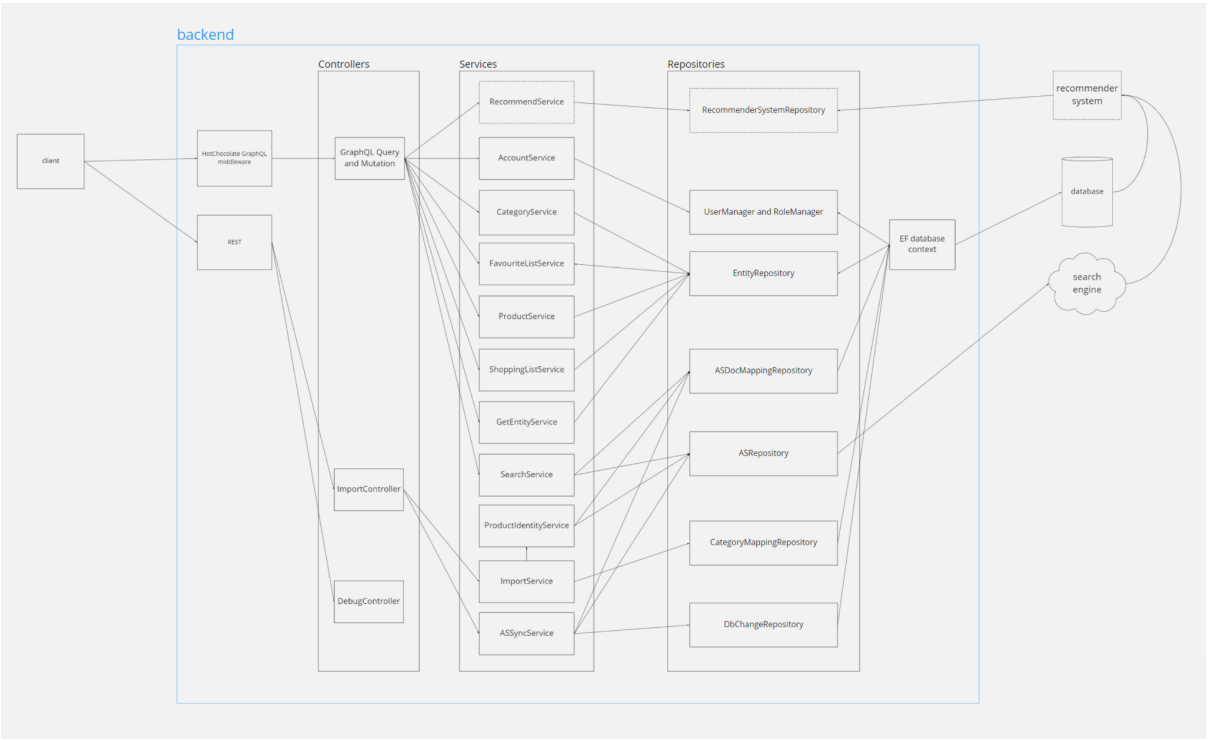
The backbone of the whole solution. As a **monolithic backend** application, it is responsible for all **data access requests** from **external client applications**. This procedure usually involves accessing the server's data sources and applying **business logic** on obtained data.

Technology

Backend app is written in **C#** using **ASP.NET Core 5** framework.
The app also utilizes **EntityFramework**, which is a ASP.NET Core library for relational database interactions (Microsoft SQL Server in our case).
Hotchocolate11 middleware and library is used to implement **GraphQL** communication with compatible client applications.

Architecture

As you can see in the diagram, the program is structured in **3 main layers** - Controllers, Services and Repositories.
Controllers are responsible for **I/O data validation and transformation**. They are also access points for clients. Typically, a controller calls a corresponding service with parameters taken from a client request and optionally transforms and returns its result.
Services execute **business logic**, using corresponding repositories.
Repositories serve as an API for **interactions** with either a database(single table) or other external component.



Database

As a database server, its primary function is **storing and retrieving data** as requested by the **backend application**.

Technology

Microsoft SQL Server is used due to its great compatibility with ASP.NET Core applications (namely EntityFramework) and developer's familiarity with relational databases. Other also suitable alternatives are MySQL, Oracle or other widely used relational databases.

Search Engine

Its primary function is to provide **full text search** capabilities. In this case it is also used to help with **product identification**.

Technology

AppSearch search engine was selected due to its **simplicity** and the possibility to have it running **locally** as an **on-premise service** for free.
Other options might include Algolia, Apache Solr or other widely used search engines.

Crawler

Its function is to **collect and send offer/product data** from online leaflets published in official websites.

Technology

The reason behind the implementation using **Python** and **BeautifulSoup4** library is for the most part its simplicity and user friendliness of mentioned crawling library. The program itself doesn't require complicated architecture therefore developing in Python makes a lot of sense.

Mobile client

The only part interacting with the **user** himself/herself.
Runs **locally** on user's mobile phone (**iOS and Android** operating systems)

Technology

React.native + Expo was chosen mainly because of its good balance between performance, development speed and developer's experience with ReactJS framework. Since the application doesn't need to have much platform specific features, native development using **Xcode or AndroidStudio wouldn't be suitable**.
App also uses **ApolloClient** library for **GraphQL** communication with the backend application and **state management** (dataflow inside an React application)
As an alternative **Flutter** would have been a good choice as well.

Code generation

- **EntityFramework** - generating model classes corresponding to database tables
- **GraphQL schema** - generating data classes corresponding to defined graphql schema (common interface) to both server and client
    - Used for data structure uniformity
    - Web and mobile clients with backend application
    - Using **graphql-codegen** tool
- **JSON schema** - generating data classes corresponding to json schema (common interface) to both server and client
    - Used for REST communication
    - Crawler with backend application
    - Using **nswag** tool (for .NET) and **datamodel-code-generator** (for Python).

Deployment

Apart from the mobile client app, every above mentioned component is running inside a **Docker container**.
Each **docker image** is created based on a **Dockerfile** specified for every component.
Containers run together managed by **docker-compose** (lightweight deployment manager).
Certain applications need to have configuration specified to run correctly. That's solved by passing **environment variables** inside **docker-compose config file** containing such values.
Data persistence is ensured by defining **volumes** (shared memory between a container and hosting machine) - database, AppSearch container.
In **actual production**, **Kubernetes** would be a better fit due to its broad feature set.