

# Milestone 2

## Implementation of a Snippet Manager

<b>Group number:</b>	42
----------------------	----

Team member 1	
<b>Name:</b>	Péter Ferenc Gyarmati
<b>Student ID:</b>	11913446
<b>E-mail address:</b>	a11913446@unet.univie.ac.at

Team member 2	
<b>Name:</b>	Simon Eckerstorfer
<b>Student ID:</b>	11911424
<b>E-mail address:</b>	simon.eckerstorfer@unet.univie.ac.at

# Overview

## Starting the App

The application can be started by executing the following command:

```
docker build . -t snip-man:nx-base && docker-compose up --build
```

A preceding build command is necessary due to the exotic nature of our development environment setup, which we discuss in detail below in the *Tech Stack* section.

The app start might take up to 10-15 minutes based on your internet connection and your machine's performance. The system will be fully functional once `snip-man-prisma-migrate` exits with code 0. Please note that it is possible that `snip-man-prisma-migrate` will exit with code 1 first, indicating an error, but this will be the case only if the mongo replica set `snip-man-mongo` starts up too slowly. The system will self-heal automatically; you should just wait until you see `snip-man-prisma-migrate` exited with code 0 in your logs.

Only two components of the system will be made publicly accessible:

- Frontend: <https://localhost>
- Backend: <https://localhost:8443/api-docs>

## Certificates

Upon system startup, a new self-signed SSL certificate is generated dynamically by executing [services/reverse-proxy/certgen.sh](#). As a consequence, you will need to allow this certificate in your browser explicitly:

- for <https://localhost:8443/api-docs>, to let the frontend access the API through <https://localhost:8443>
- for <https://localhost>, to let you visit the web application.

Alternatively, you can set your browser to ignore all certificate errors.

## Technology Stack

We decided to use a monorepo to develop the project in order to be able to share code between the web and server modules. We used [Nx](#) as the base of our workspace, which handles all of the build processes as well as the dependency interleavings between the modules. This is the reason why the application startup requires two docker commands:

- `docker build . -t snip-man:nx-base` is responsible for packing the whole monorepo with all development dependencies installed into a single docker image, called `snip-man:nx-base`. The root-level [Dockerfile](#) is used to create this base image.
- `docker-compose up --build` is responsible for building the individual components and starting them. The root-level [docker-compose.yml](#) file is used for this step.

## Information System Components

In what follows, we give an overview of the components that make up our information system.

### Relational Database

- [PostgreSQL](#) version 14.2
- Docker image: `postgres:14.2`
- Local Dockerfile: [services/postgres/Dockerfile](#)
- Service name in [docker-compose.yml](#): `postgres`

### NoSQL Database

Please note that we are running MongoDB as a single-node replica set in order to provide support for database transactions, as it is required by Prisma, the ORM we are using.

- [MongoDB](#) version 4.2.10
- Docker image: `bitnami/mongodb:4.2`
- Local Dockerfile: [services/mongo/Dockerfile](#)
- Service name in [docker-compose.yml](#): `mongo`

## Object-Relationship and Object-Document Mapper

We decided to make use of [Prisma](#) to define and handle our database schemas both for the relational and non-relational parts of our project. It comes with its own lightweight schema-definition language, based on which the structure of both databases can be defined automatically. In addition, it made it possible for us to leverage generated code to interact with our postgres and mongo components in a type-safe manner.

- [Prisma](#) version [3.14](#)
- Docker image: snip-man:nx-base
- Local Dockerfile: [Dockerfile](#)
- Service name in [docker-compose.yml](#): prisma-migrate

The prisma-migrate service is responsible for applying the schema defined in [services/postgres/schema.prisma](#) and [services/mongo/schema.prisma](#) to the postgres and mongo components as soon as they are started.

## Backend

We picked NestJS for our backend framework due to its TypeScript support, well-defined architecture as well as its elaborate support for creating REST APIs. Due to the fact the framework builds on top of Express.js, it is capable of everything that Express could accomplish, but it eliminates some of the more tedious low-level tasks it comes with.

Please note that we strived to follow best practices when it comes to backend development; therefore, we documented the REST API we exposed using the industry-standard OpenAPI. It is available for observation under <https://localhost:8443/api-docs>.

We mostly used [Faker](#) to populate the database with randomised data. Additionally, we used the GitHub API to get some real Tag names and copied code snippets from various sites and our own VSCode collection.

- [NestJS](#) version 8.0.0, OpenAPI
- Docker image: snip-man:nx-base
- Local Dockerfile: [apps/server/Dockerfile](#)
- Service name in [docker-compose.yml](#): server

## Frontend

For the web application development, our decision fell to Next.js due to the pleasant developer experience it provides as well as the advanced optimisations it does under the hood.

In order to create a user interface which is not only functional but also aesthetically pleasing, we utilised TailwindCSS for low-level design, and we leveraged Geist UI components for higher-level UI details.

We utilised vanilla React primitives such as contexts and reducers for state management, as well as we pulled in React Query as a dependency to facilitate the interaction with the REST API our server component exposes.

- [Next.js](#) version 13.9.6,
- State management:
  - [React](#) version 17.0.2 (Contexts & Reducers)
  - [React Query](#) version 3.39.1
- UI Libraries:
  - [Geist UI](#) version 2.3.8
  - [TailwindCSS](#) version 3.1.2
- Docker image: snip-man:nx-base
- Local Dockerfile: [apps/web/Dockerfile](#)
- Service name in [docker-compose.yml](#): web

## Reverse Proxy

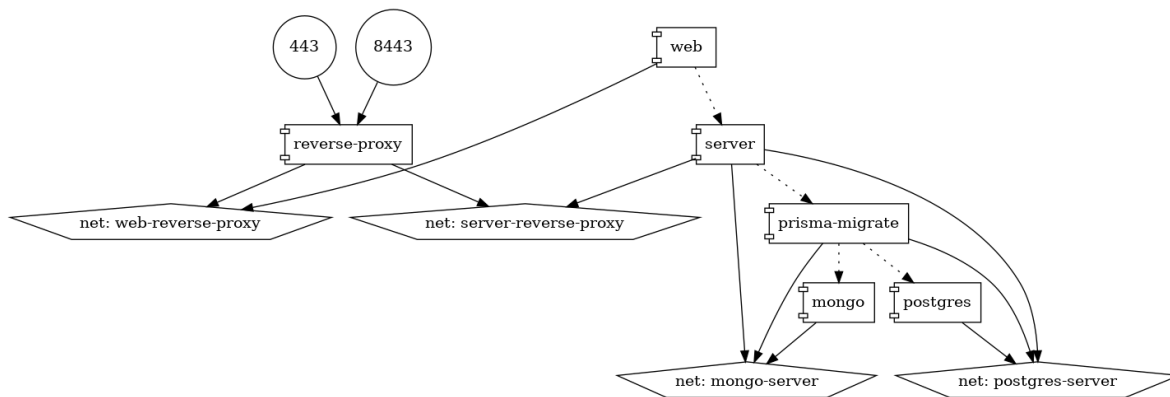
To keep our application secure and to make sure that we expose only those components to the outside world which are absolutely necessary, we took advantage of Nginx. We added support for automatic self-signed SSL certificate generation using a custom shell script: [services/reverse-proxy/certgen.sh](#).

- [Nginx](#) version 1.21
- Docker image: nginx:1.21
- Local Dockerfile: [services/reverse-proxy/Dockerfile](#)
- Service name in [docker-compose.yml](#): reverse-proxy

The service exposes the following components:

- web: <https://localhost>
- server: <https://localhost:8443/api-docs>

## Docker Services Overview



The component diagram above visualises the defined Docker networks, services as well as the dependencies between them. As apparent, the databases are the services which need to be accessible the soonest, as all the other components depend on them. Please note that - as the network topology also testifies - we tried to follow the principle of least privilege when orchestrating the services. Therefore, we allow communication between components on dedicated networks only when it is actually sensible that they have a link with each other.

Had we not used this fine-granular approach and put all components on the same network, the web component would be able to access both databases directly, the database servers would be able to communicate with each other, and the reverse proxy would also have access to components it does not need to know anything about.

Also, as it can be seen in the visualisation, the only entry points to the information system are through ports 443 and 8443. The web application and the backend can be reached respectively through these ports via HTTPS.

The diagram above was generated from our root [docker-compose.yml](#) file using an open-source library called [docker-compose-viz](#).

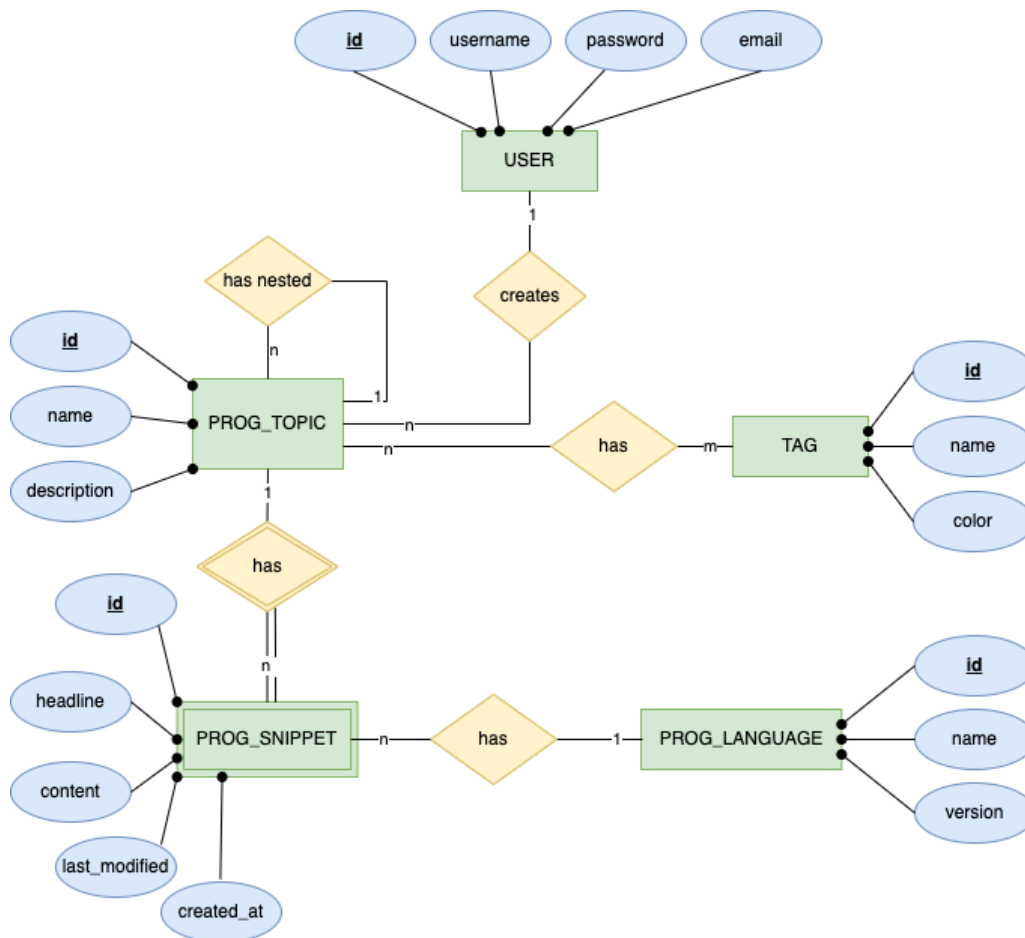
# Database Design Decisions

We dedicate this section to giving a detailed overview of the database design decisions we made throughout the project. We will discuss the design of the SQL and NoSQL components, our considerations when it comes to migrating the relational schema to a document-based one, and we will provide a comparison of the queries relevant to the use cases we defined.

## SQL Design

### Relational ER Design

Based on the feedback we received after submitting the first project milestone, we adapted our Entity-Relationship diagram to reflect the relationship between PROG\_SNIPPET and PROG\_LANGUAGE entities properly.



## Physical Design

As discussed in the *Tech Stack* section, we utilised Prisma to define the base schema for our databases; however, in what follows, we are presenting the actual SQL script generated by the tool. It is notable that a unique index for `user.email` and `user.username` got generated, as we know that they are used to identify users without any ambiguity.

Another aspect of the physical design worth mentioning is the definition of `prog_snippet` as a weak entity: the fact that it possesses a composite primary key, depending on a parent `prog_topic` makes it evident that no instance of `prog_snippet` can exist without an enclosing `prog_topic`.

Finally, please note that the optional recursive (zero-to-many) relationship on `prog_topic` entities is implemented using a nullable foreign key `parent_id`.

```
1 -- CreateTable
2 CREATE TABLE "user"
3 {
4   "id"      TEXT NOT NULL,
5   "email"   TEXT NOT NULL,
6   "username" TEXT NOT NULL,
7   "password" TEXT NOT NULL,
8
9   CONSTRAINT "user_pkey" PRIMARY KEY ("id")
10 };
11
12 -- CreateTable
13 CREATE TABLE "prog_topic"
14 {
15   "id"      TEXT NOT NULL,
16   "name"     TEXT NOT NULL,
17   "description" TEXT NOT NULL DEFAULT E'',
18   "parent_id" TEXT,
19   "user_id"  TEXT NOT NULL,
20
21   CONSTRAINT "prog_topic_pkey" PRIMARY KEY ("id")
22 };
23
24 -- CreateTable
25 CREATE TABLE "tag"
26 {
27   "id"      TEXT NOT NULL,
28   "name"     TEXT NOT NULL,
29   "color"    TEXT NOT NULL,
30
31   CONSTRAINT "tag_pkey" PRIMARY KEY ("id")
32 };
```



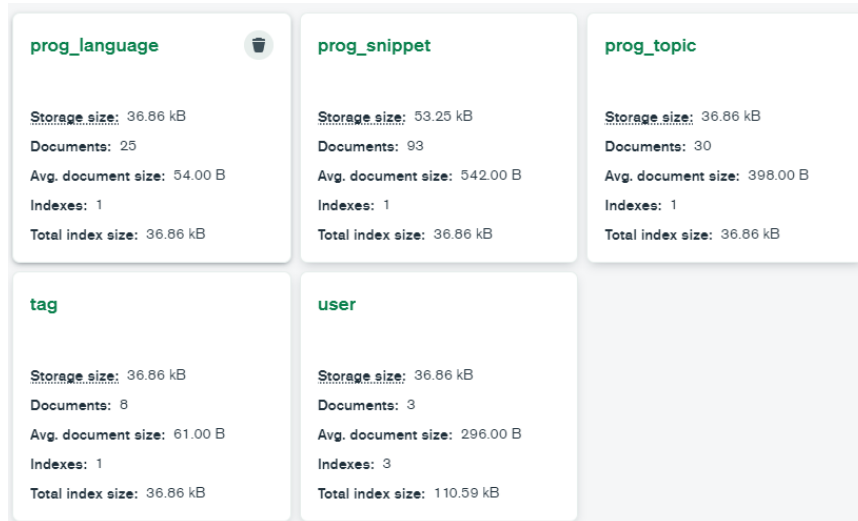
```
33
34 -- CreateTable
35 CREATE TABLE "tags_on_prog_topics"
36 (
37   "tagId"          TEXT NOT NULL,
38   "progTopicId"    TEXT NOT NULL,
39
40   CONSTRAINT "tags_on_prog_topics_pkey" PRIMARY KEY ("tagId", "progTopicId")
41 );
42
43 -- CreateTable
44 CREATE TABLE "prog_snippet"
45 (
46   "id"              TEXT          NOT NULL,
47   "headline"        TEXT          NOT NULL,
48   "content"         TEXT          NOT NULL,
49   "created_at"       TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
50   "last_modified"    TIMESTAMP(3) NOT NULL,
51   "prog_topic_id"    TEXT          NOT NULL,
52   "prog_language_id" TEXT          NOT NULL,
53
54   CONSTRAINT "prog_snippet_pkey" PRIMARY KEY ("id", "prog_topic_id")
55 );
56
57 -- CreateTable
58 CREATE TABLE "prog_language"
59 (
60   "id"              TEXT NOT NULL,
61   "name"            TEXT NOT NULL,
62   "version"         TEXT NOT NULL,
63
64   CONSTRAINT "prog_language_pkey" PRIMARY KEY ("id")
65 );
66
67 -- CreateIndex
68 CREATE UNIQUE INDEX "user_email_key" ON "user" ("email");
69
70 -- CreateIndex
71 CREATE UNIQUE INDEX "user_username_key" ON "user" ("username");
72
73 -- AddForeignKey
74 ALTER TABLE "prog_topic"
75 ADD CONSTRAINT "prog_topic_user_id_fkey"
76 FOREIGN KEY ("user_id") REFERENCES "user" ("id")
77 ON DELETE CASCADE ON UPDATE CASCADE;
78
79 -- AddForeignKey
80 ALTER TABLE "prog_topic"
81 ADD CONSTRAINT "prog_topic_parent_id_fkey"
82 FOREIGN KEY ("parent_id") REFERENCES "prog_topic" ("id")
83 ON DELETE CASCADE ON UPDATE CASCADE;
84
```

```
85 -- AddForeignKey
86 ALTER TABLE "tags_on_prog_topics"
87 ADD CONSTRAINT "tags_on_prog_topics_progTopicId_fkey"
88 FOREIGN KEY ("progTopicId") REFERENCES "prog_topic" ("id")
89 ON DELETE CASCADE ON UPDATE CASCADE;
90
91 -- AddForeignKey
92 ALTER TABLE "tags_on_prog_topics"
93 ADD CONSTRAINT "tags_on_prog_topics_tagId_fkey"
94 FOREIGN KEY ("tagId") REFERENCES "tag" ("id")
95 ON DELETE CASCADE ON UPDATE CASCADE;
96
97 -- AddForeignKey
98 ALTER TABLE "prog_snippet"
99 ADD CONSTRAINT "prog_snippet_prog_topic_id_fkey"
100 FOREIGN KEY ("prog_topic_id") REFERENCES "prog_topic" ("id")
101 ON DELETE CASCADE ON UPDATE CASCADE;
102
103 -- AddForeignKey
104 ALTER TABLE "prog_snippet"
105 ADD CONSTRAINT "prog_snippet_prog_language_id_fkey"
106 FOREIGN KEY ("prog_language_id") REFERENCES "prog_language" ("id")
107 ON DELETE SET NULL ON UPDATE CASCADE;
108
```

## NoSQL Design

Similarly to SQL, we defined the schema using Prisma. We created five collections, which correspond to the SQL tables with the exception of the bridging table “tags\_on\_prog\_topics”.

You can see a screenshot of the collections from MongoDBCompass below.



Collection	Storage size	Documents	Avg. document size	Indexes	Total index size
prog_language	36.86 kB	25	54.00 B	1	36.86 kB
prog_snippet	53.25 kB	93	542.00 B	1	36.86 kB
prog_topic	36.86 kB	30	398.00 B	1	36.86 kB
tag	36.86 kB	8	61.00 B	1	36.86 kB
user	36.86 kB	3	296.00 B	3	110.59 kB

Again, as mentioned in the SQL section, indices for `user.email` and `user.username` were generated, in addition to the `id` indices.

The actual documents for each collection are shown in their respective section below, together with reasoning for the NoSQL design decisions we made and screenshots of an exemplary document taken from MongoDBCompass.

### User

```
_id: ObjectId('62ab09fb89987f389fc218c5')
email: "Laverne_Carroll71@hotmail.com"
username: "Laverne_Carroll71"
password: "Kp4qWYmHlf0zlkG"
prog_topic_ids: Array
  0: ObjectId('62ab09fb89987f389fc218c8')
  1: ObjectId('62ab09fb89987f389fc218c9')
```

User documents are relatively similar to their SQL counterparts. They are realised in their own collection, as they are read frequently on the home page.

In general, topics are always read with regard to their respective user (never globally). We considered embedding them but refrained from it due to the fact that topic documents have nested documents themselves and can get large (discussed below). Therefore, we only hold an array of topic IDs.

## Programming Topic

```
_id: ObjectId('62ab09fb89987f389fc218c9')
name: "transmit"
description: "Use the digital SQL array, then you
user_id: ObjectId('62ab09fb89987f389fc218c5')
tags: Array
  0: Object
    name: "timeline"
    color: "#522a07"
  1: Object
  2: Object
parent_id: ObjectId('62ab09fc89987f389fc218e0')
prog_snippet_ids: Array
  0: ObjectId('62ab09fc89987f389fc218f1')
```

Programming topic documents show a greater difference from their corresponding SQL table.

The first consideration was handling the relationship to the snippets. We decided to save an array of IDs of related snippets instead of embedding the snippets themselves. This was decided due to the fact that snippet content is not limited in size; thus, the snippet objects may get relatively big.

When it comes to the nesting of topics themselves, we also decided against embedding them, as the nesting depth is not limited. Our use-case is to show a topic tree in the UI and create sub-topics. We achieved this by only including the parent ID of a topic - root topics do not have a parent ID. When adding a sub-topic, existing topics do not need to be updated, as the new sub-topic only needs to include the parent ID.

The embedding of tags will be discussed in the next paragraph.

## Tag

```
_id: ObjectId('62ab09fc89987f389fc218e6')
name: "svelte-components"
color: "#18771a"
```

Generally speaking, tags are tiny documents that are read frequently but written rarely. As they are read in conjunction with topics most of the time, they are embedded within the topics themselves (see above).

The set of tags also needs to be queried as a whole for the language dominance report. Therefore, tags also have their own collection to provide an efficient query.

## Programming Snippet

```
_id: ObjectId('62ab09fc89987f389fc218f1')
headline: "Select database table and populate"
content: "$result = $mysqli->query('SELECT * FROM students');
        while ($row = $res..."
created_at: 2022-06-16T10:44:47.334+00:00
last_modified: 2022-06-16T10:44:47.335+00:00
user_email: "Laverne_Carroll71@hotmail.com"
prog_topic_id: ObjectId('62ab09fb89987f389fc218c9')
✓ prog_language: Object
  name: "PHP"
  version: "8.0"
```

Even though snippets are weak entities and are only used together with topics, it is not sensible to embed them into topic documents for the reasons discussed above. The topic ID is used to create an efficient relation.

As far as programming languages go, they are tiny documents and are needed for each reading of the snippet document; thus, we embedded them into the snippets.

The user email address is also embedded into this document to increase the efficiency of the query for the first report, as we spare a join operation that is necessary when using SQL queries.

## Programming Language

```
_id: ObjectId('62ab09fb89987f389fc218ac')
name: "C++"
version: "20"
```

Similar to tags, programming languages are tiny documents that are read frequently but written rarely.

As mentioned in the snippet paragraph, they are embedded there to increase efficiency (no join operation needed). They also have their own collection (also similar to tags) to be able to efficiently query them when selecting a language (when creating a snippet and selecting a language for the first report).

## Database Queries

In this section, we discuss, compare and contrast the queries utilised to accomplish the defined use cases and reports.

### Preface

As already outlined, we used Prisma as an ORM. As a result of this, queries were generated automatically for the use cases. Yet, we will discuss the produced code too in order to demonstrate our understanding of the underlying fundamental concepts.

On the other hand, as the upcoming sections testify, we wrote the queries for the reports ourselves in order to attempt to maximise their performance.

We used a convenient, declarative, annotation-based library called [class-validator](#) for the input data validation on the server-side (in addition to validating input on the client-side as well). The annotated Data Transfer Objects are located in [libs/entities](#).

## Use Case: Create a Topic

Author: Péter Ferenc Gyarmati

*A user can create a new topic with a name and description in order to better categorize code snippets. A topic can be created as a root topic as well as a subtopic inside existing topics.*

The SQL and NoSQL statements for inserting a new programming topic are fairly similar. A notable difference between them is that the NoSQL statement involves inserting an empty array for the tags assigned to the topic. This is due to the fact that tags are embedded to prog\_topic documents in MongoDB, whereas the many to many relationship is modelled with an auxiliary table in PostgreSQL; therefore, no information about the tag entity can be seen when it comes to the insertion of a topic. Please note that the tags array is empty due to the fact that this use case does not involve assigning tags to a topic.

### PostgreSQL

```
1 INSERT INTO "public"."prog_topic"
2 ("id", "name", "description", "parent_id", "user_id")
3 VALUES
4 ($1, $2, $3, $4, $5)
5 RETURNING "public"."prog_topic"."id"
```

### MongoDB

```
1 db.prog_topic.insertOne({
2   name: "NestJS & Swagger",
3   description: "Integration of OpenAPI standards with NestJS",
4   parent_id: ObjectId("62a8bb1c5e70b5d630876cc0"),
5   user_id: ObjectId("62a8bb1c5e70b5d630876cb9"),
6   tags: []
7 });
```

## Report: Find users active in specific languages

Author: Péter Ferenc Gyarmati

*This report enumerates the email addresses of all users who have created at least 3 code snippets in the last month written in a specific programming language.*

### PostgreSQL

1. Select all `prog_snippet.id`, `prog_snippet.prog_topic_id` tuples where the programming language id matches the passed `progLanguageId` parameter
2. Join the `prog_topic`, `user` and `relevant_snippets` (created in the previous step as an in-memory alias) relations to access the filter criteria. By starting the join from the `prog_topic` we can make sure that even the snippets created inside nested topics will be considered.
3. Group by user email address and keep only those rows in which the count of relevant snippet ids is at least 3, using the `HAVING` construct.
4. Order the result set by email address (the only column we have)

```
1 --- Initial filtering for snippets with given lang and timestamp
2 WITH relevant_snippets AS (
3     SELECT id, prog_topic_id
4     FROM prog_snippet
5     WHERE prog_language_id = ${progLanguageId}
6     AND created_at BETWEEN (now() - INTERVAL '1 month') AND now()
7 ),
8 --- After filter, perform join and aggregation to access the relevant user info
9 report_result AS (
10    SELECT "user".email AS user_email
11    FROM prog_topic
12    JOIN "user" ON "user".id = prog_topic.user_id
13    JOIN relevant_snippets ON relevant_snippets.prog_topic_id = prog_topic.id
14    GROUP BY user_email
15    HAVING COUNT(relevant_snippets.id) >= 3
16 )
17 --- Order the result set by the email address
18 SELECT user_email
19 FROM report_result
20 ORDER BY user_email;
```



## MongoDB

1. Project fields of `prog_snippet` to keep only those which are needed for the criteria check and define `prog_snippet_created_at` as a time ISO string to make it possible to compare `ISODate` objects in step 2
2. Filter for those `prog_snippet` documents which have the specified language name and their creation falls in the range of the last month. Please note that the query purposefully does not consider language version, as the report aims to collect user email addresses who are competent in a specific language in general, regardless of its version.
3. Group by email, aggregate the count
4. Keep only those users' email addresses who authored at least 3 snippets matching the above-defined conditions
5. Keep only the email field
6. Sort by email address

```
1 db.prog_snippet.aggregate([
2   // 1. Keep only the relevant fields, get the creation date as ISO string
3   {
4     $project: {
5       _id: 1,
6       prog_language_name: "$prog_language.name",
7       prog_snippet_created_at: {
8         $convert: { input: "$created_at", to: "string" }
9       },
10      user_email: 1
11    }
12  },
13  // 2. Keep only the relevant snippets
14  {
15    $match: {
16      $expr: {
17        $and: [
18          { $eq: ["$prog_language_name", progLanguageName] },
19          { $gte: ["$prog_snippet_created_at", oneMonthAgo] },
20          { $lte: ["$prog_snippet_created_at", now] }
21        ]
22      }
23    }
24  },
25  // 3. Group by email, assign it to doc _id, as it is unique at this stage
26  { $group: { _id: "$user_email", count: { $sum: 1 } } },
27  // 4. Keep only those emails where the snippet count is at least 3
28  { $match: { $expr: { $gte: ["$count", 3] } } },
29  // 5. Keep only the email field, without the count
30  { $project: { _id: 0, email: "$_id" } },
31  // 6. Sort by email
32  { $sort: { email: 1 } }
33 ]);
```

## Comparison

The MongoDB query should be more efficient than its PostgreSQL counterpart due to the fact that we designed the document-based collections with use case performance in mind. By embedding the programming language and the user email address into the snippet documents, we spare the explicit joins which we have in the SQL statement.

Both queries are optimised in the sense that they are performing projection and filtering before data joining, this way reducing the number of rows to be present in the aggregation phases of the execution.

## Indexing Possibilities

When it comes to enumerating the email addresses of all users who have created at least 3 code snippets in the last month written in a specific programming language, `prog_snippet.created_at` plays a crucial part in the query. It might be reasonable to create a time-to-live index on the field to enhance the lookup performance. Under the hood, a dedicated thread would be running periodically. It would scan the `prog_snippet` collection for documents with an expired TTL, and they would be removed in the background in a way such that it is totally transparent for the developer. This way, we could prevent an ever-growing index in our database.

## Use Case: Add a New Snippet to a Topic

Author: Simon Eckerstorfer

*A user can create a new code snippet with a headline and content. The associated programming language and version has to be selected from a list of possibilities.*

Again, the two queries are relatively similar; they only differ in two points, which were mentioned in the NoSQL Design part already: The `user_email` is additionally included in the MongoDB document to increase efficiency, and the `prog_language` is embedded instead of only including the FK to spare a join operation.

One additional important thing to mention is the need to update the `prog_topics` collection when adding a snippet in MongoDB. As the `prog_topics` hold an array of related snippet IDs, this array of given `prog_topic_id` needs to be updated.

*Note: Opposed to what was said in M1, the version cannot be selected separately here. This does not change the data model, however.*

## Postgres

```
1 INSERT INTO "public"."prog_snippet"
2 ("id","headline","content","created_at","last_modified","prog_topic_id","prog_language_id")
3 VALUES
4 ($1,$2,$3,$4,$5,$6,$7)
5 RETURNING "public"."prog_snippet"."id", "public"."prog_snippet"."prog_topic_id"
```

## MongoDB

```
1 db.prog_snippet.insertOne({
2   headline: "Copy list by value",
3   content: "final newList = [ ...oldList ];",
4   created_at: DateTime("2022-06-16 15:12:52.672 UTC"),
5   last_modified: DateTime("2022-06-16 15:12:52.672 UTC"),
6   user_email: "Daija_Aufderhar93@hotmail.com",
7   prog_topic_id: ObjectId("62ab47db886bf4c59c74a492"),
8   prog_language: { name: "Dart", version: "2.1", },
9 });
```

## Report: Find the most dominant languages by tags

Author: Simon Eckerstorfer

*This report enumerates the programming languages in which the most lines of code snippets have been written under a specific tag.*

Note: For simplicity reasons, instead of the number of “lines of code”, the number of characters is counted in the report.

Also, the result is limited to 10 to have a good overview.

### Postgres

1. All relevant topics for the given tag id are selected
2. Snippets that are included in those relevant topics are selected, and a column for the length of their content is created.
3. Programming languages are joined by id to get their name and version. The joined table is grouped by the languages and a column is created with the aggregated sum of the included snippets. They are then ordered by the length and take the first 10.

```
1 --- 2. Select snippets that are in relevant topics
2 --- and create a column for the length of their content
3 WITH relevant_snippets AS (
4     SELECT prog_language_id, LENGTH(content) as local_len
5     FROM prog_snippet
6     WHERE prog_topic_id IN
7 --- 1. Select relevant topics for given tag id
8         (SELECT progTopicId
9          FROM tags_on_prog_topics
10         WHERE tagId = ${tagId}))
11 --- 3. Join with prog language and group snippets by language
12 --- create an aggregate column with the sum of the grouped snippets
13 --- Order by the length and take the first 10
14 SELECT name, version, SUM(local_len) as length
15 FROM relevant_snippets
16 JOIN prog_language
17     ON prog_language.id = relevant_snippets.prog_language_id
18 GROUP BY prog_language_id, name, version
19 ORDER BY length DESC
20 LIMIT 10
```

## MongoDB

1. Starting from the topic collection, only take topics which include the given *tagName*
2. Only keep programming snippet id array, which is needed for following lookup
3. Lookup programming snippets that are associated with the filtered topics
4. Only keep the snippets; ids are not needed anymore
5. Unwind the arrays of snippets into a single document for each snippet
6. Only keep the programming language and the content of the snippet to increase efficiency
7. Group by language as this is required by the report. Also, create an aggregate *length* field with the sum of the length of the content of each grouped snippet
8. Project into wanted output format
9. Sort by the aggregate length field. If the length is equal, sort by name
10. Only take the first ten results.

## Comparison

While the MongoDB query with its nine stages seems more complicated, it is arguably easier to reason with and more explicit in what it does.

Even though optimisations were done during the NoSQL schema creation, the query does need a lookup operation (for reasons discussed above at the Programming Topic NoSQL Design - snippets are too big to embed them).

The programming languages are embedded in the snippet document, thus not needing to be joined (as opposed to the SQL query).

In general, it was a point of consideration to only work with the rows and columns needed, making the queries more efficient. You can explicitly see this in the MongoDB query, where match and project are used extensively, in particular before the lookup.

```
1 db.progTopic.aggregate([
2   {
3     // 1 .filter topics with the given tag name
4     $match: {
5       tags: {
6         $elemMatch: { name: tagName },
7       },
8     },
9   },
10  {
11    // 2. only keep snippet id arrays
12    $project: {
13      _id: 0,
14      prog_snippet_ids: 1,
15    },
16  },
17  {
18    // 3. join snippets that are associated with the topics
19    // as prog_snippet_ids is an array, prog_snippets is also an array (of snippets)
20    $lookup: {
21      from: 'prog_snippet',
22      localField: 'prog_snippet_ids',
23      foreignField: '_id',
24      as: 'prog_snippets',
25    },
26  },
27  {
28    // 4. only keep snippet arrays
29    $project: {
30      _id: 0,
31      prog_snippets: 1,
32    },
33  },
34  {
35    // 5. create a document for each snippet
36    $unwind: '$prog_snippets',
37  },
38  {
39    // 6. keep only the relevant fields
40    $project: {
41      lang: '$prog_snippets.prog_language',
42      content: '$prog_snippets.content',
43    },
44  },
45  {
46    // 7. group by language
47    // create an aggregate length field with the sum of the length of the content
48    $group: {
49      _id: '$lang',
50      length: {
51        $sum: { $strLenCP: '$content' },
52      },
53    },
54  },
55  {
56    // 8. map fields to create wanted output format
57    $project: {
58      _id: 0,
59      name: '$_id.name',
60      version: '$_id.version',
61      length: 1,
62    },
63  },
64  {
65    // 9. sort by length descending and name ascending
66    $sort: {
67      length: -1,
68      name: 1,
69    },
70  },
71  {
72    // 10. only take the first 10 results
73    $limit: 10,
74  },
75 ],);
```