POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Software development life cycle of a back-end web application using the Elixir language

Supervisor

Antonio Servetti

Candidate

Rizwan Khalid

Edoardo Conte (Company Supervisor)
Restworld S.r.l.

July 2023

Abstract

In the Horeca sector, finding the right candidate is a problem that every employer face, and the same goes for people who look for work in this sector. This situation has worsened in COVID-19 where finding people to come and work in the restaurant became difficult. Restworld has a solution for this, giving a platform to both the worker and employer to enlist their requirements, working for specific time-period / duration, or for long term. Among many other platforms providing this service, in the Restowrld, the focus is the correct match between the two entities, simplicity and ease-of-use product for them. Certainly this includes a reliable and a scalable product. The backend of the software is developed in Elixir language which is known for its robustness, reliability and scalability. The elixir backend is connected to a relational database, which ensures the consistency by having some constraints, like unique or foreign-key constraints. Every software platform involves the authentication and authorization mechanisms developed. Considering that fact the some users in this domain are not much into technology, they tend to forget their passwords frequently. Magic link functionality was developed so that any user, given email can access the platform, without the need of a password. The user experience matters a lot in software products and if the backend system is slow in providing the data, this can cause some significant delays in rendering it on client and thus making your service bad. To tackle this issue, cache has been implemented for some functions which we know are used very frequently. To provide quick and results from a number of documents, or any data, full-text search has been implemented by using an external service. The only thing is to have consistent data on both our database and the external service servers. By having these implementations in the product, we advantages in different contexts. For the main users, they get to access the platform without remembering almost anything. The system responds quickly as the processing time reduced for some mainly used operations. The internal members gain more flexibility in finding a worker by remembering a specific thing about them, like their city or a nearby area.

Acknowledgements

First of all, I am sincerely grateful to God, for His boundless blessings and unwavering presence, which have bestowed upon me the strength and perseverance to successfully complete this thesis. Without His guidance and grace, this milestone would not have been attainable.
I am deeply appreciative of my family for their unwavering love and unconditional support during this significant journey. I am especially grateful to my parents, whose continuous encouragement and sacrifices have been instrumental in my pursuit and successful completion of this endeavor.
I am also indebted to my supervisor, Prof Antonio Servetti, for his guidance and expertise. I am so grateful for his support and encouragement.
I am deeply indebted to Restworld for providing me with the opportunity to conduct my research. I could not have completed this thesis without their help. Specifically, I would like to thank Edoardo Conte, my supervisor at Restworld, for his support and guidance during my time at the company. He was always available to answer my questions and provide me with feedback.

Table of Contents

List of Tables

List of Figures

Acronyms

Horeca
: Hotels, Restaurants and Catering

API
: Application Programming Interface

REST
: REpresentational State Transfer

Chapter 1

Introduction

1.1 Software Development Life Cycle

Software Development Life Cycle (SDLC) is a process through which defines some steps for a software system. This process helps organization in creating an efficient and client adhering system, by also managing the risks and control the cost of the system.

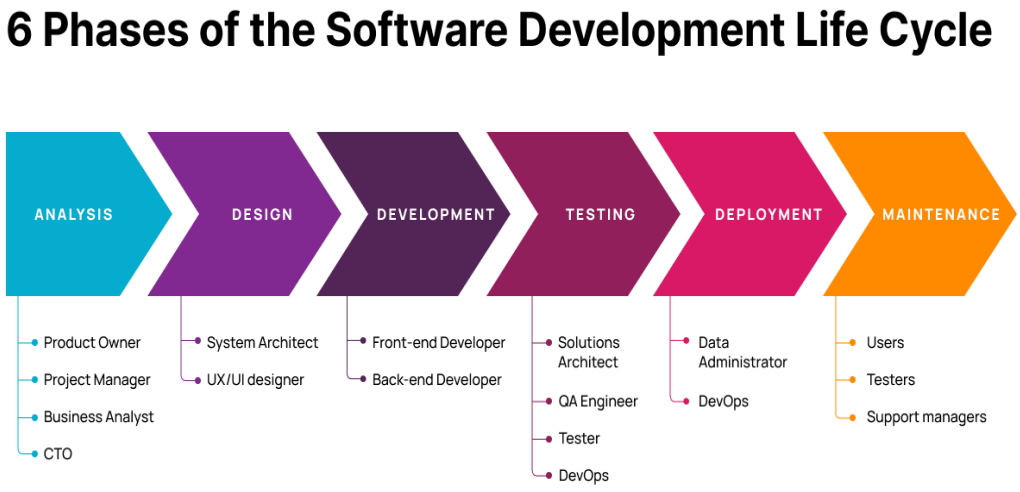

Figure 1.1: Software Development Life-Cycle

Figure 1.1 outlines the steps involved, along with some responsible/associated people to it. We can have a brief introduction to each step. [1]

- **Requirement**
  This is the initial phase for development life cycle, in which we analyze the situation, gather the requirements for the system. The goal is to have definite features and constraints of the software. This process requires the involvement of different stakeholders, business analysts and development teams.

- **Design**
  This step involves designing the architecture. Once we have the requirements ready, we can start defining the database structure and the UX/UI design. Defining how the software would look like, the process flow for user and how it will be stored in the database is all done in this step.

- **Development**
  The actual coding takes place at this development step. The right language and tools for this system is defined in this step. This step involves programming the software according the specifications defined already.

- **Testing**
  Once we have a software programmed, we can pass it to the testing phase in order to verify that the requirements are met. Critical features, bugs are all identified in this step. QA engineers, testers are responsible for the verification process.

- **Deployment**
  After testing the software, making sure that it satisfies the requirements, its ready to be deployed for the end-users. An executable is made and is installed on to production environment.

- **Maintenance**
  The last step is to monitor the deployed software, in order to fix the bugs or having any unusual activities. Several monitoring tools are available. Feedbacks are highly important in this step

1.2 Problem Description

For a startup company, there are many challenges faced in terms of launching their product/service in the market. Here we can briefly look at some them, related to a software product.

- Due to **limited** financial **resources**, startups usually have difficulties in hiring professional designers, developers in their team. This can affect the overall product development.
- Having close deadlines, **time restrictions**, to release the Minimum Viable Product (MVP) into the market, which would eventually generate cashflow, this causes the company to make some strategic decisions and prioritize the software features.
- There can be intense **competition** in the market, considering that there already exists established companies with greater resources available, it's difficult to achieve the customer loayalty.

1.3 About Restworld

Restworld is a startup company that provide services in Horeca sector by simplifying the supply-demand gap. It acts as a bridge between restaurant owners (employers), who are looking for someone to hire, for full-time or seasonal jobs, and the workers who are looking for work in this sector. A dedicated team of Customer Success Managers (CSM) is there to understand the needs and requirements for the employers and workers. Each employer can have multiple retaurants and for each restaurant, there can be different job positions. A brief flow of work can be described as,

- The employers conveys the requirements to the CSM for the new job position
- The responsible CSM finds a worker fit enough for the profile described by the employer
- Once a list of workers, that satisfies the requirements, are ready, that shortlisted workers/candidates are sent to the employer
- The employer can have interviews and hire among those shortlisted candidates

The company has provided different platforms for each entity to access their services. An employer has a profile/dashboard available in which there are shortlists available for each job position and can also see the list of shortlisted workers.
In the same manner, the workers have a separate platform in which they can see their status in shortlists or can apply to other active job positions directly.
Then there is an internal platform which is used by CSMs to look thtough profiles and find the candidates.

Chapter 2

Tools and Technologies

There can be many solutions to a given problem, the selected solution should be effective given the scenario. In the same way, developing a software product, we have multiple ways to follow, many tools and technologies at hand. Listed below are some of the technologies that were used.

2.0.1 Database Design

Structuring the database is useful in order to have complete view of the system. We can analyze whether all the requirements are met or not. This step is crucial for a system because it can happen that the proposed structure cannot be scalable in future.

For creating the structure, there are many tools available such as [2],

- Diagrams.net
- Dbdiagram
- ERD Plus
- QuickDBD

dbdiagram was used to define our database structure, in which we can define the tables, their columns and the relationships between them. It's a free and simple tool for creating Entity Relationship Diagrams.

Dbdiagram uses DBML (Database Markup Language), which can be seen on left in figure 2.1. We just to define in the table schema, the references it has and the tool will create the reference in the diagram shown. It also has a real-time collaborative editing feature.

Figure 2.1: Dbdiagram

2.0.2 Project Management

Project management is essential in a software development teams as they provide a centralized system to creating tasks, keeping track of them, respecting deadlines and seeing who is falling behind. This helps in making the development organized. Brief advantages of a project management tool are,

- **Accountability/Responsibility**: A task can be assigned to a person and he will be repsonsible that the task satisfies the requirements. There are also stories which defines the high-level requirements, then the tasks can be created according to what needs to be done to implement the functionality
- **Visibility**: Being able to see all the tasks of the project, the user can look at the risks involved, the complexity of each task and the dependencies of it, which allows the user to make better decisions.

Some of the tools for project management are,

- Jira
- ClickUp
- Basecamp

Jira is used by us for project management, as it has good integrations with other tools, highly customizable and widely used.

2.0.3 Communication

In order to make the team effective, communication plays a big part in it. It ensures that everyone is on the same page, have the requirements clear to everyone and convey if there any deadlocks or hurdles that might help the productivity.
Communication can be done in many types, it can be,

- Face-to-Face
- Calls
- Email
- Messaging
- Video Calls

Slack and Notion are used as modes of communication. Slack supports all the modes of communication, also it has variety of apps/extensions available through which we can integrate other applications to it. For example, we can integrate Jira directly with Slack and we will know the updates on the tasks to which we have been subscribed to.
Notion on the other hand, is more of a documentation tool. Let's say we have an idea about the feature that can be or is to be implemented, we create a notion page which can be shared among users, they can collaborate with each other and improve the productivity. Having a documentation of different features of a software is important and can be guideline for new users in the team.

Chapter 3

Implementation

3.1 Database Architecture

Database architecture defines the desgining and organization of a database, how the components are structured and the relationships that exists between them. The decision in choosing the data model for the database is crucial as this will define overall efficiency, scalablity and integrity of a system.
The data model that we are using is the Relational model, Relational Database Management System (RDBMS). The relational model has several advantages, such as, [3]

- Strict foreign key constraints, these constraints help in maintaining consistency all over the system. If we have different user-types exist as schemas, then we are sure that the user record exists for that user-type entity.
- Structured data, each entity type having a structure of its own, such as mandatory (not null), optional (nullable) fields and even having enum types of a field. As shown in figure 3.1, we can see the different user-types, enum values, that exists in the system.
- Indices, by defining the indices on specific columns helps us in making the database system effective.

3.1.1 Brief Database Structure

In figure 3.2, the main schemas defined in our backend system is shown. User schema has one-to-one relations with Workers, Employers and RwMembers, each schema containing its own specific fields. The Employer can have many restaurants, so there exists a one-to-many relation between Employers and Restaurants, and similarly, a

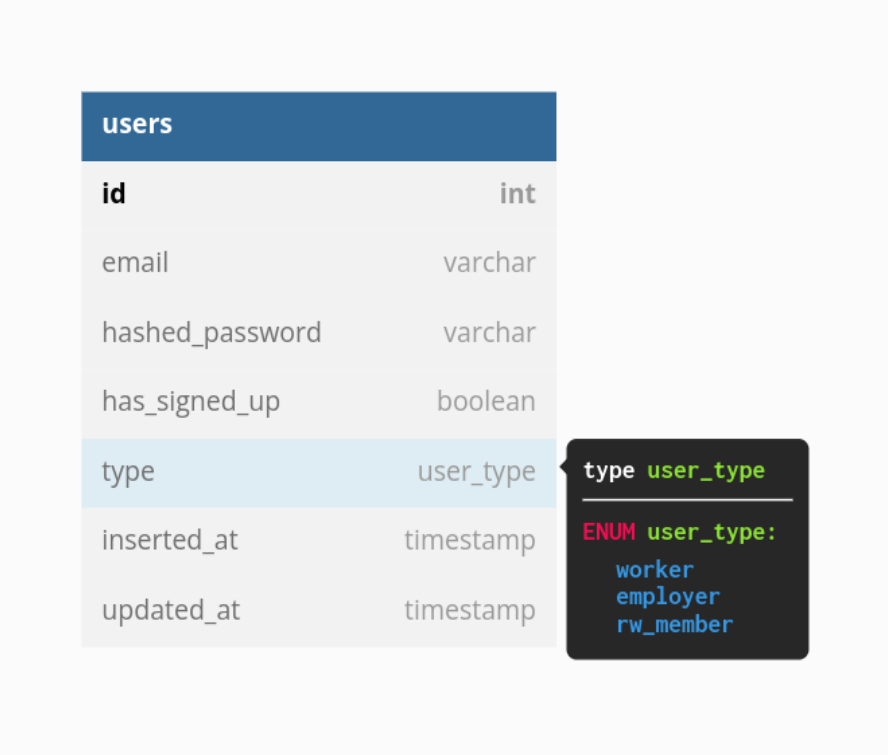

Figure 3.1: User Schema

Restaurant can have multiple Job positions, it has a one-to-many relation with the Job Positions. Variables table contains data related to information about different entities, such as some job_type of a job position, gender required for a job position, etc. The Variables contain the key and variable_name, as a unique multi-column. The label field is used to display the text that will be visible to the user, while in the backend system, we use the key or the ID to reference them, so that if we want to change the label in future, the existing reference with the entities will not change.

3.1.2 Our Implementation

There are many referenced entities and these references, with time, will increase. There will be more joins between schemas, only to get the information. Giving preference to efficiency and scalablity in our case, we disregard the strict foreign key constraints for some entities. Lets take an example of Variables schema. Because the values of Variables are change rarely, and when it does, we can propagate the changes manually, we store the variables data in memory. Now the variable-referenced-fields are simple strings and doesn't have a foreign key constraint on

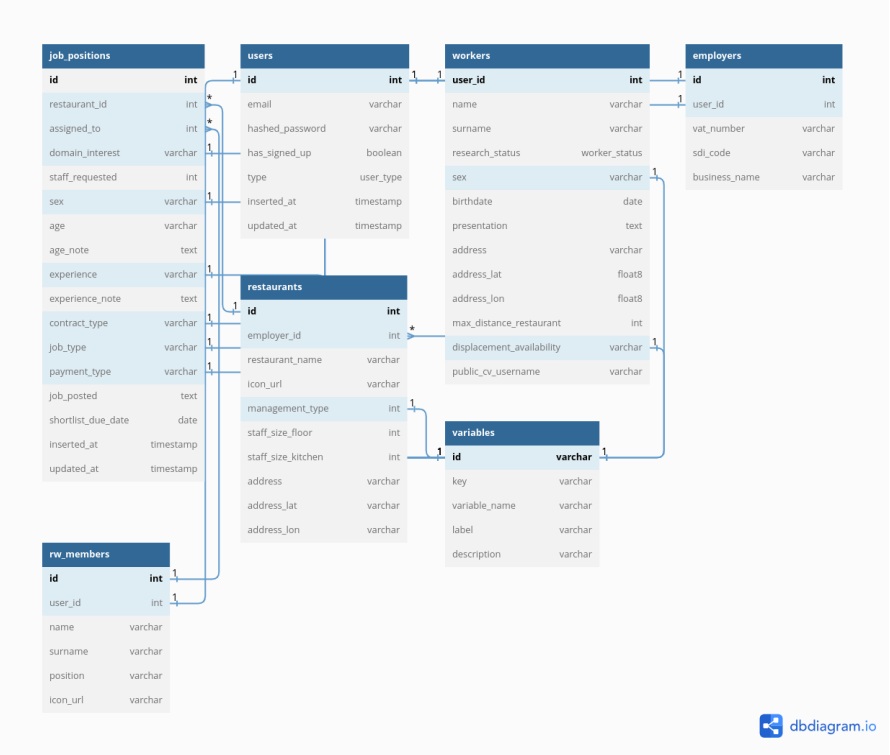

Figure 3.2: Database Structure

Database. For example, the job_type field of a job position is a string now, which contains the value of the Variable IDs but there doesn't exist a constraint between them. In this way, we can decrease the number of joins because we fetch the information from memory, which is much faster.

3.2 Programming Language

The backend system is written in Elixir language. Elixir is a functional programming language built on Erlang virtual machine, BEAM which is known for its concurrency, fault-tolerance and scalablity. Discord can be seen as an example of this language's usage, which scaled their system to have 5 million concurrent users.[4]
Here are some of the key features of Elixir:

- **Fault Tolerant**: Elixir has been designed to be scalable and fault tolerant. It uses actor model for concurrency and isolates the processes. Each process can have a supervisor which is responsible for its functionality to be completed, or in case of failures, gives the restart ability or a recovery process.
  Here is an example of a supervised task (caching a value). The last option,

restart: :transient makes sure that this task is restarted specified number of times, only in the case of failure.

```
Task.Supervisor.start_child(
  RestWorld.TaskSupervisor,
  fn ->
    :ok = Cache.put({@key_atom, jp_id}, jp, ttl: @ttl)
  end,
  restart: :transient
)
```

- **Concurrency**: Elixir processes are very lightweight and not to be be confused with operating system processes. They communicate with each other through message passing, are isolated and share nothing, making them fault tolerant and its usage in a distributed system.
- **Immutability**: Elixir emphasizes immutability and the using pure functions. Function arguments are transformed but cannot be modified. Add it to the pattern matching and the code becomes maintainable.
  Here in an example, we have a **sum** function which will be executed only if the **num1** and **num2** are integers and returns the sum of them. In the function body, if the value of either of them changes, it doesn't propagate outside the scope of the function.

```
def sum(num1, num2) when is_integer(num1) and
  is_integer(num2) do

  num1 + num2
end

def sum(num1, num2), do: :error
```

- **Syntax**: Elixir is inspired by Ruby. It provides concise, readable and gives features like pattern-matching, pipelines and macro-based programming.

3.3 Authentication and Authorization

Authentication is verifying that the user is the one who he/she claims to be, to grant access to a system. To authenticate a user, we use password-based authentication and magic link functionality.

- **Password Based**: The user is authenticated using the email and password, which stored in our database. Hashed value of password is stored. When the user logs in, a JWT (JSON Web Token) is generated and assigned to user. Then whenever the user makes the subsequent requests, that JWT token is passed in header of the request, and the backend identifies the identity of the user through it.

- **Magic Link**: It's a much simpler login functionality for a user. The process is as follows:

  - User provides an accessible email
  - The backend generates a token, encrypting user's identity, and sends it to the user's provided email
  - The user opens the link in his/her email and is redirected to login page
  - The login function at backend decrypts the token, extracts the user identity, generate and returns the JWT token

Authorization comes after the authentication step which gives privileges to user and access to contents to which it has been allowed. We use authorization to protect content of other users. Authorization process makes it possible to not allow a user1 see the contents of user2.
Bodyguard library is used at backend for authorization. It allows us to define different scenarios to authorize a user. Lets have an example,

```
def authorize(actions, {:ok, %{type: "employer"} = _user}
 = _current_user, _params)
    when actions in [
           :index_restaurants,
           :index_job_positions,
           :retrieve_invoice,
           :upload_picture,
           :delete_picture,
           :stripe_customer_portal
         ],
    do: :ok

    # deny everything else
    def authorize(_, _, _) do
      {:error, :unauthorized}
    end
```

authorize function defined above ensures that only the user with type = employer can access the functions, which are listed as **actions**. For every other action or if the conditions are not met, the user is denied access

3.4 Cache

3.4.1 Need for cache

The relational database is used in the system. To get information for a single entity, we need to create joins on database, depending on what information do we need. Lets take an example,

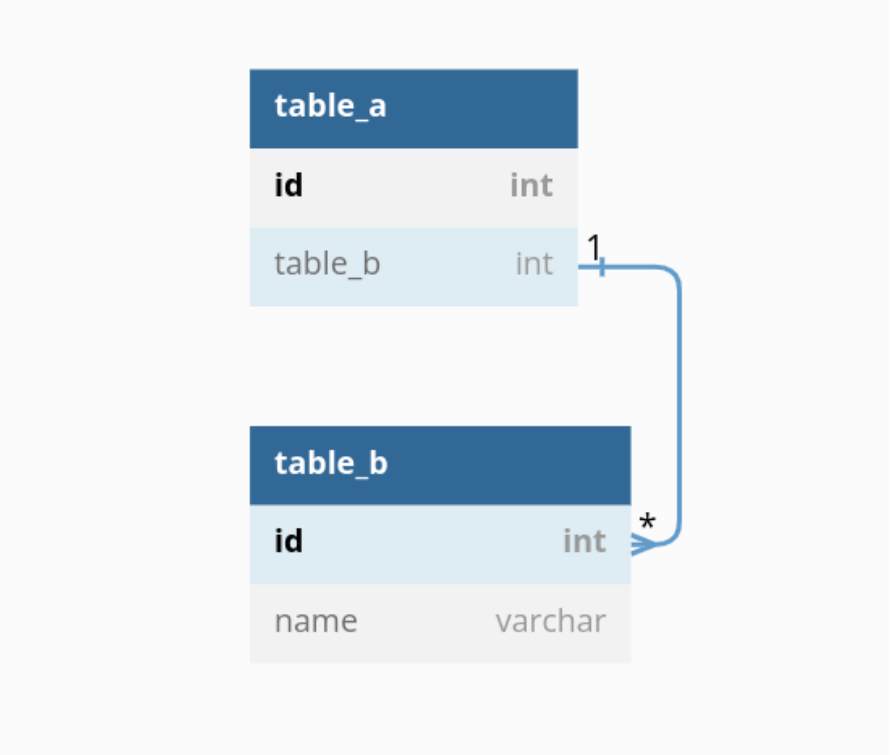

Figure 3.3: Referenced Tables

Here we can see that the *table a* has one-to-many relation with *table b*. In order to get information for *table b* while querying *table a*, we need to create the join which will load multiple *table b* values.
Now imagine a case where we have multiple of these tables and these joins have to be created everytime the enpoint is called, that would definitely overload the database. In order to solve this efficiency problem, cache has been implemented,

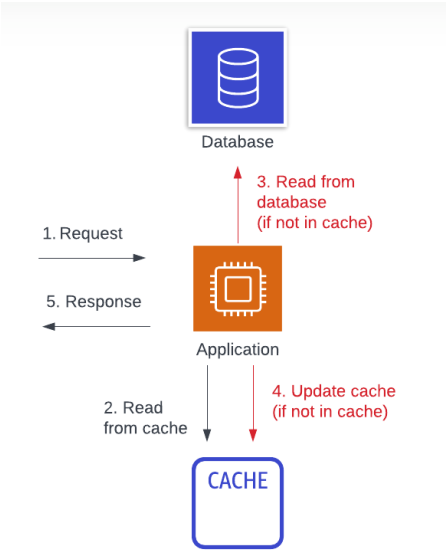

Figure 3.4: Read Aside

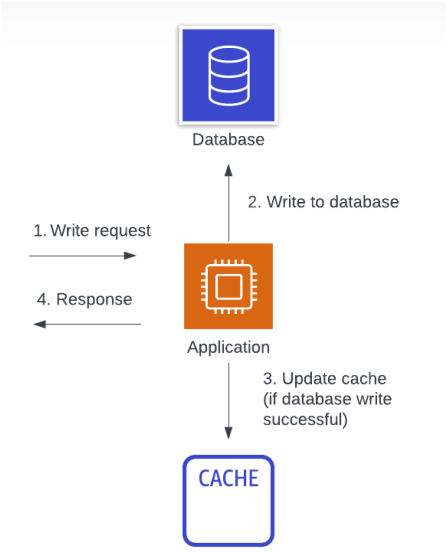

Figure 3.5: Write Aside

which would *cache* the values, and the next time that operation is called, the cached value is returned rather than going to the database.

3.4.2 Caching Patterns

Cache Aside

Cache aside strategy is the most commonly used one. Whenever there is a read request, the application will first look in cache, if the value is not present, the application will load the value from database and update the value in cache.
For a write request, the application writes to the database first and then to the cache.

Read/Write Through

In this cache pattern, the cache is used as a primary SOR (Source of Record). In these cases, rather than the application, the cache is responsible for providing you the data. If there is a read request and the data is not present in cache, the cache is responsible for getting the data from the data-store and return the value.
For the write request, the application conveys the request to the cache directly and the cache is responsible to update the data on data-store. In a way, in this usage pattern, the cache is the orchestrator here and is responsible to maintain the data consistency between cache and data-store.

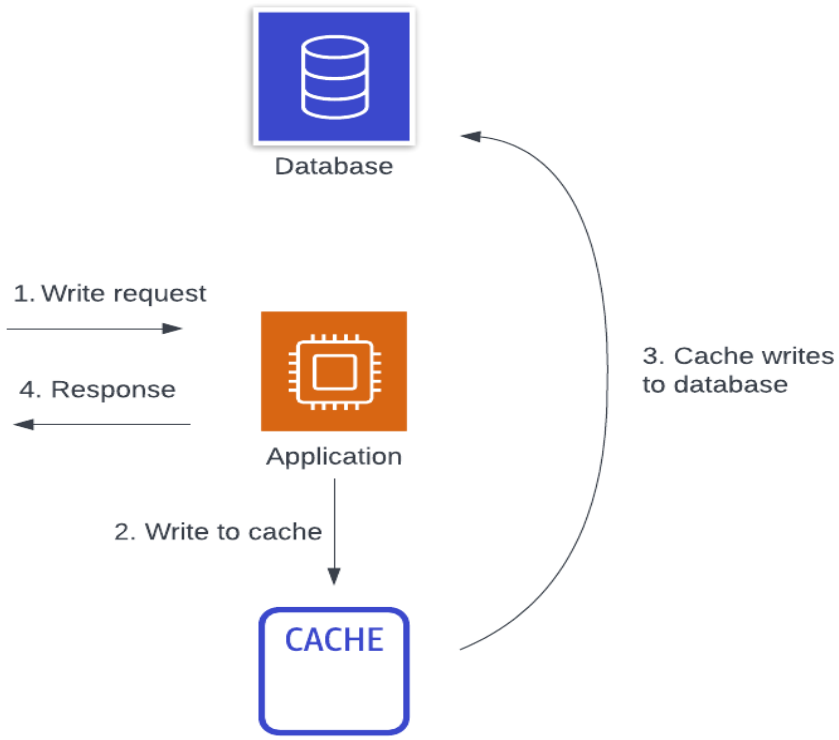

Figure 3.6: Write Through

Write Behind

For the write-behind usage pattern, the difference exists with the cache updating the data-store. With the write-through strategy, a synchronous request is made to update the data-store while for the write-behind strategy, an asynchronous request is made to update the value in data-store.

3.4.3 Caching Topologies

- **Replicated Cache**
  In this cache topology, the key-values are replicated for each node, that is, each node/server will contain the same key-value pairs. In terms of reading, this would be very useful but a write will trigger the write operation on every node. Therefore, this kind of topology is useful for cases in which the data is read most of the times and is written less often or only once.

- **Partitioned Cache**
  For the partitioned cache topology, the key-value pairs are distributed among each node/servers. In order to get a value, difference algorithms are used,

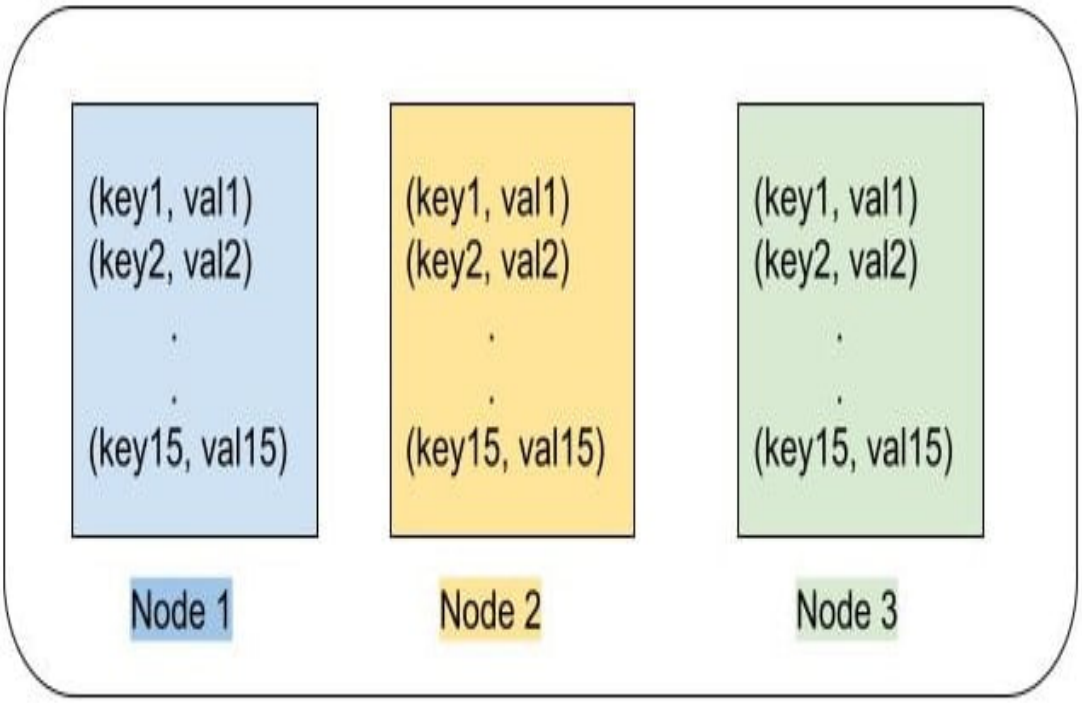

Figure 3.7: Replicated Cache

such as, consistent hashing, to find the node containing the requested value.

3.4.4 Our Implementation

For our implementation of cache, considering that a simple solution would be suffice and which would be easily maintainable, cache-aside pattern is implemented.
Regarding the topology, both the topologies are implemented depending on the use case. For some tables/entities having bulk data, partitioned/distributed cache topology is used while for other tables having less data but frequently used, replicated cache topology is implemented.

In Figure 3.9, we can see an example of a large size information. A single job position table has its own fields but if we need information also from its restaurant, the data size will be increased. Also these entities are to be updated more frequently, therefore it is more sensible to put them in a distributed cache.

As small sized information, shown in Figure 3.10, the information contained in this table is rarely or never updated, therefore a replicated cache is a better choice

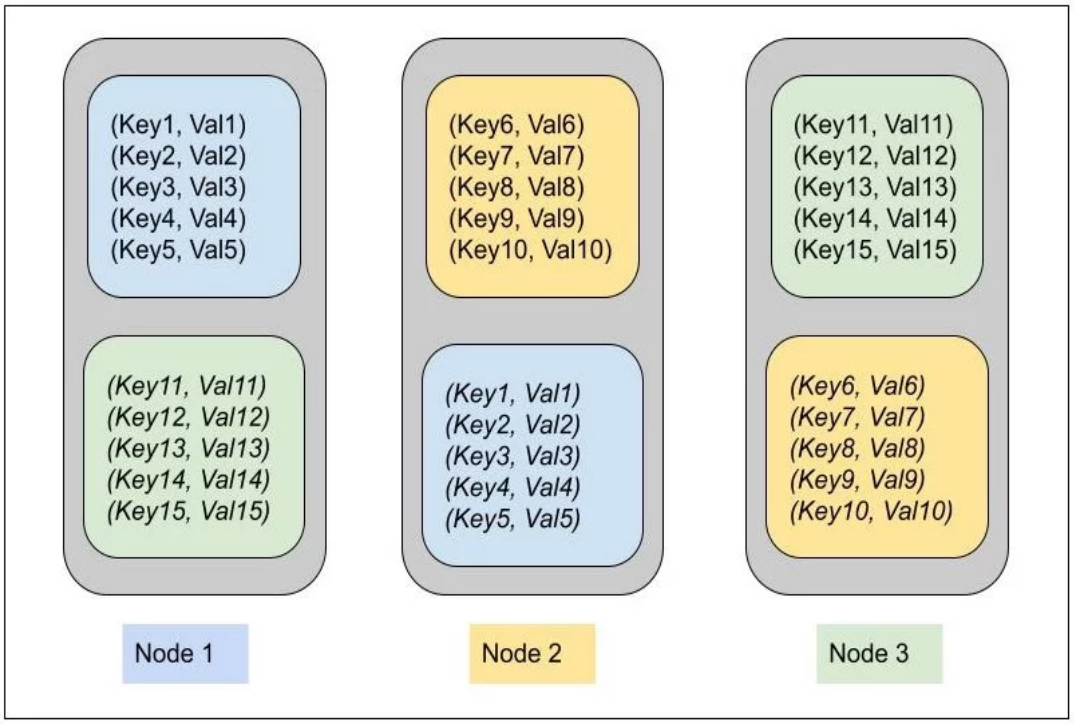

Figure 3.8: Partitioned Cache

to implement them.

In the code snippet given, listing 3.1, a wrapper function is defined, which we call to store the job position in cache. The code that calls the library function to save the job position, is wrapped inside a supervised task. A supervised task is a task which will be restarted by the supervisor if its execution is not normal, that is, an error or exception occurs.

Listing 3.1: Storing a Job Position in Cache

```
1  def put(%JobPosition{id: jp_id} = job_position) do
2    Task.Supervisor.start_child(
3      RestWorld.TaskSupervisor,
4      fn ->
5        :ok = Cache.put({@key_atom, jp_id}, job_position, ttl: @ttl)
6      end,
7      restart: :transient
8    )
9
```

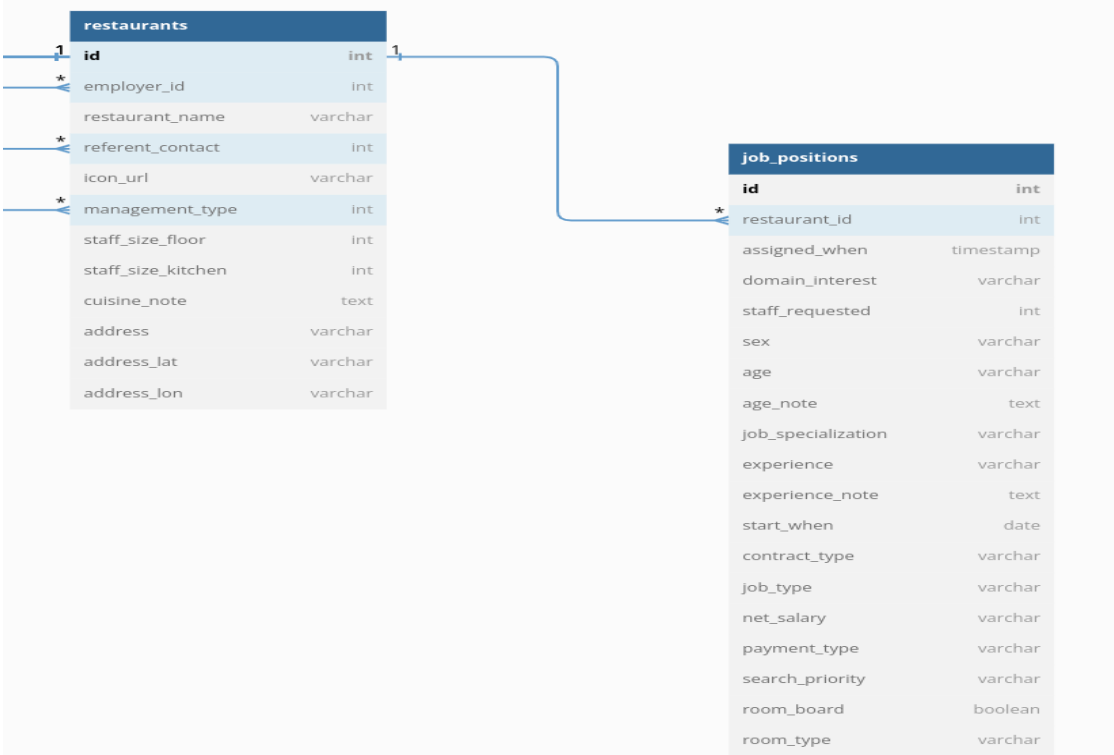

Figure 3.9: Large Sized Table

```
10      job_position
11    end
```

Requesting Job Position (when the value doesn't exist in Cache)

- Job position is requested given the ID
- System checks value in cache
- Value not found in cache
- Database is queried for that ID
- Joining operations are done to preload the entities
- Preloading job position is stored in cache
- Job position is returned to client

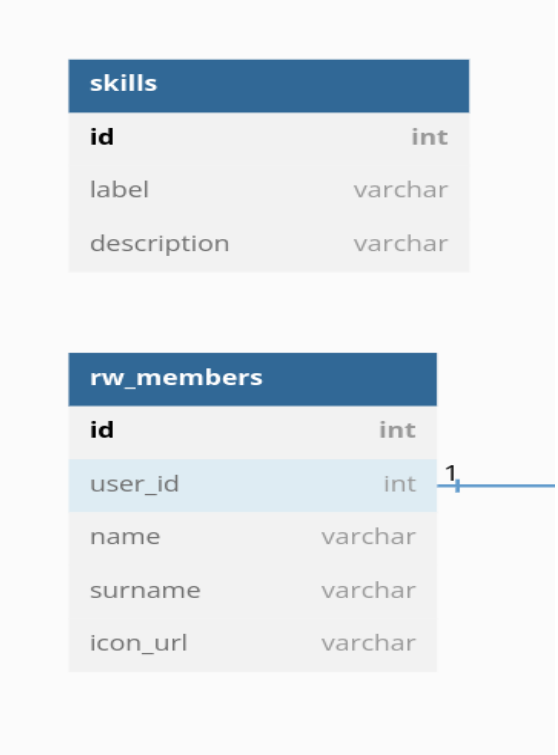

Figure 3.10: Small Sized Tables

Listing 3.2: Fetching a Job Position

```
def get(JobPosition = entity, %{"id" => id}) do
  Cache.get(entity, id)
  |> case do
    nil ->
      RestWorld.Employers.Helper.get_job_position_from_db(id)
      |> case do
        nil ->
          nil

        jp ->
          Cache.put(jp, true)
      end

    job_position ->
      job_position
  end
end
```

Updating a Job Position

- PUT request is made given the ID and update parameters
- Update is performed in Database (making DB as the SOR)
- Updated value is stored in cache
- If the value was already present in cache, it is replaced with the updated value
- All other related cache values are deleted from cache, so that when the request that value is made, updated value from cache is queried and cache is updated for that value

Listing 3.3: Updating a Job Position

```
def update(changeset) do
  Repo.update(changeset)
  |> case do
    {:ok,
     %JobPosition{restaurant_id: restaurant_id} =
         updated_job_position} ->
      JobPosition.upsert_algolia_index(updated_job_position)

      Cache.put(updated_job_position)
      Cache.delete(%Restaurant{id: restaurant_id})

      {:ok, updated_job_position}

    error ->
      error
  end
end
```

In the code snippet above, updated job position value is put in cache. Now, since the restaurant values, stored in cache, also has job positions preloaded, that restaurant, associated with the job position, is deleted from cache, otherwise the data in cache would be inconsistent.

Table 3.1: Cache vs DB - Speed Difference

Entity	Size (bytes)	DB Read (msec)	Cache Read (msec)
JobPosition	38028	59973	731
Restauarant	208031	26258	2813
RWMember	4493	1541	33

3.5 Full Text Search Implementation

Full text search is used to retrieve information from an entire text of a document, or from the data containing phrases, keywords. It is used to query the text data given the input. A document can be described as a collection of phrases, text or keywords, to retrieve the information from.
Lets understand the concept of full-text search with an example. Imagine a person was reading a book, now that person needs to find a quote, the location of that is not known to the person. What the person only remembers is that the quote contained 'blue' in it. Using this limited information, the user can get the result using full text search implementation.
To efficiently retrieve the information, an index is created for each keyword/phrase, with its location or an object. For example, we have 3 objects, as,

```
{
  id: 1,
  published_date: "2001-01-02",
  text: "The sky was filled with mesmerizing blue clouds."
},
{
  id: 2,
  published_date: "1996-05-02",
  text: "As I gazed up, the blue clouds danced gracefully."
},
{
  id: 3,
  published_date: "2005-02-02",
  text: "The moonlight shimmered on the tranquil lake."
}
```

The indices for the objects can be,

```
'sky' => 1,
'blue' => 1, 2,
'filled' => 1,
'gracefully' => 2,
'shimmered' => 3
```

The indices contain the phrases and the id of the object. Therefore, searching for 'blue' would return 1 and 2 objects.

3.5.1 The need for full-text search

The main entities defined in our system are of workers, restaurants and job positions. In each of these entities, there are some text/string columns defined. Lets say in one

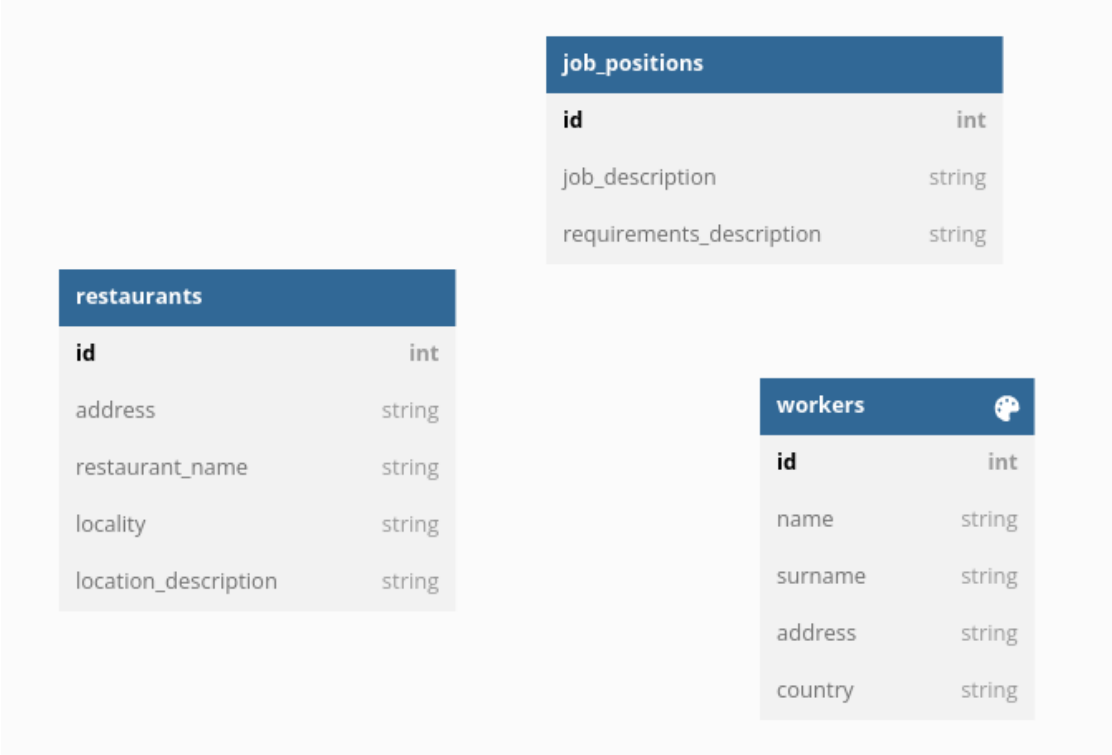

Figure 3.11: String Fields Example

of the restaurants location_description, we had stored near metro station. Rather than remembering the name of the restaruant and querying only the restaurants table, we can query the pharse, metro station, which would return all the restaurants having the 'metro station' in its location_description.
This searching is useful also for other entities making the work of the user a lot easier. Since the full-text search is not only limited to text, we store whatever information we want and query on that.

3.5.2 Algolia

Algolia offers its search engine through Software-as-a-Service (SaaS) model. It allows developers to integrate the relevant search functionality into their platforms. It includes many of the features, like autocomplete, instant search, typo tolerance, etc. Algolia focuses on speed, simplicity and relevance to enhance their search

functionalities. RESTful API is used for searching.

Implementation

Algolia requires that the information to be uploaded on their servers. Multiple indices are created on Algolia for different entities. There is a separate index for workers and another one for restaurants, so that if we need information about the workers only, we can use that index only and filter the unnecessary results.

Listing 3.4: Restaurant Algolia Structure

```
def restaurant_struct(restaurant) do
  %{
    objectID: restaurant.id,
    restaurant_name: restaurant.restaurant_name,
    cuisine_note: restaurant.cuisine_note,
    address: restaurant.address,
    country: restaurant.country,
    locality: restaurant.locality,
    postal_code: restaurant.postal_code,
    notes:
      Enum.map(restaurant.notes, fn note ->
        %{
          objectID: note.id,
          note: note.note,
          category: note.category,
          deleted: note.deleted,
          last_modified_by: note.last_modified_by_user
        }
      end),
    job_positions:
      Enum.map(restaurant.job_positions, fn jp ->
        %{
          objectID: jp.id,
          job_description: jp.job_description,
          status_key: jp.status_key
        }
      end),
    employer: %{
      objectID: restaurant.employer.user_id,
      business_name: restaurant.employer.business_name,
      vat_number: restaurant.employer.vat_number,
      sdi_code: restaurant.employer.sdi_code
    }
  }
end
```

In the code given, we have an example struct defined for restaurants. This object is uploaded to the algolia server with the information filled. Whenever we have

an insert or an update operation to a restaurant value in database, the value on algolia is updated/inserted synchronously.
Algoliax library is used to communicate with Algolia. Helper functions like save, search and many others are called for different purposes. These functions make the API request to algolia, get the result from Algolia API, parses it and returns to the user. Using a library, it has been integrated in our codebase, which handles the insertion and update of the data on Algolia.

Usage

On algolia, limited information of an entity is stored. Because the information that is left out is not needed in any case for searching.
Since the data is not complete over aloglia, we cannot query only algolia and return the results to the client. In our implementation, in case where we want to search for a given text, both algolia and the DB is queried in a specific manner.

Text-Search Filtering:

- Query algolia for text search query
- Apply the filters on Algolia, if present any
- Get the entity IDs from algolia
- Query the DB with those those IDs returned
- Preload the necessary information
- Maintain the same order as from algolia
- Return the results

Basic Filtering:

- Query the DB with the filters present in parameters
- Preload the necessary information
- Apply sorting
- Return the results

Chapter 4

Testing and Maintenance

4.1 Test-Suite

In order to maintain the functionalities of a system, test suite is essential. It makes sure that each of the feature is working correctly after addition of some another functionality or refactoring a current one. Instead of testing each feature manually which takes time, tests are created for that feature which would save time and increase productivity.
In elixir, we have a built-in framework for testing, ExUnit, which allows to create tests in a very readable manner. Lets take a look at the following example,

```
test "create_user()" do
  user_attrs = %{
    "name" => Faker.Person.name(),
    "email" => Faker.Internet.email(),
    "type" => "worker",
    "password" => Faker.Lorem.characters(16)
  }

  assert {:ok, %User{} = user} =
    Accounts.create_user(user_attrs)

  assert user.name == user_attrs["name"]
  assert user.email == user_attrs["email"]
  assert user.type == user_attrs["type"]
  assert Bcrypt.verify_pass(user_attrs["password"],
     user.hashed_password)
end
```

In the test module, we define a test named *create_user*. In the test, we define

attributes to create a user and pass it to the context function. Now if everything is correct, the user is created with the given attributes, that we assert afterwards with the given attributes.

4.1.1 Our Implementation

The tests are run synchronously, unless specified otherwise, but in a random order. Now imagine a test which would need to test the get/show function of an entity. For that we would need to create user in each test and then assert that the value returned from the context function is the correct one, as defined below,

```
test "get_user()" do
  user_attrs = %{
    "name" => Faker.Person.name(),
    "email" => Faker.Internet.email(),
    "type" => "worker",
    "password" => Faker.Lorem.characters(16)
  }

  {:ok, %User{} = created_user} =
    Accounts.create_user(user_attrs)

  assert {:ok, user} = Accounts.get_user(created_user.id)
  assert user.id == created_user.id
end
```

The same criteria would be used for other functions which involves a user entity to be present. To avoid having to create user everytime in each test, we are defining the main entities in a **setup_all** function which would run once at the start of each testing module. Each test can select the entity it needs, as implemented below,

```
setup_all do
  user_attrs = %{
    "name" => Faker.Person.name(),
    "email" => Faker.Internet.email(),
    "type" => "worker",
    "password" => Faker.Lorem.characters(16)
  }

  {:ok, %User{} = created_user} =
    Accounts.create_user(user_attrs)
```

```
  {:ok, user: created_user}
end

test "get_user()", %{user: user} do
  assert {:ok, user1} = Accounts.get_user(user.id)
  assert user1.id == user.id
end

test "update_user()", %{user: user} do
  update_attrs = %{age: 33}
  assert {:ok, updated_user} =
    Accounts.update_user(user, update_attrs)
  assert updated_user.age == update_attrs.age
end
```

Now the setup_all function returns the entities that can be used by the function. Each test, after its name is being defined, it selects/accepts the entities that it is required. The point to remember here is that if we change an entity/update a shared entity, it will modify it for the other test as well. In that case, we will just create a bare minimum entity, which will be explained below.

Using Factories

Factories are the entities which are created using the minimum information. For example we can create a user using just email and type, name and password are not required for creating a user. ExMachina is used to create factories for each entity in test suite. Lets take an example of the definition of a user factory,

```
def user_factory do
    %User{
      email: Faker.Internet.email(),
      type: "worker",
      password: Faker.Lorem.characters(19)
    }
  end
```

Now our test module can import the factories defined and use them as,

```
...after importing the factory module
test "get_user()" do
  user = insert(:user)
  assert {:ok, user1} = Accounts.get_user(user.id)
```

```
    assert user1.id == user.id
  end
```

So instead of creating an entity, containing bulk information which is not necessary in every case, we can create the entities with the minimum required information, we can create them easily and use them.

4.2 Monitoring

When the application is deployed to production, in order to ensure that the it is working as expeted, a or multiple monitoring tools can be implemented. They are responsible to report any irregularities found with the application. Since the error code 500, internal server error, happens on unknown cases, it's crucial to know when and where it happened, and if it's frequent, then it needs to be prioritized over others.

4.2.1 Appsignal

In our system, Appsignal has been integrated, as a monitoring tool, which gives us many useful insights into the system. Whenever there is a system crash, we can see the log report generated by the library on the dashboard. Appsignal also allows to monitor custom functions, calculating their execution time and differentiating the events that are heppening during its execution.
For the general usage of appsignal, we can simply configure the file for it and use it in our main application. The config file allows us to ignore some actions or to filter some parameters that we don't want to be logged or sent to the appsignal. For example, due to privacy concerns, the user's address, phone numbers, passwords, etc, are not sent to appsignal.

As seen for the dashboard figure, 4.1, we have multiple tools available for monitoring or to diagnose a problem. One of them is the Slow Queries functionality. It lists all the queries that sorted by impact or throughtput, which helps us to give an idea of what can be optimized, whether we can optimize the query or if we can cache it. We can see the details like which query is actually executing, what is it's response time, to which event it's associated. Here is a screenshot of the slow queries section.

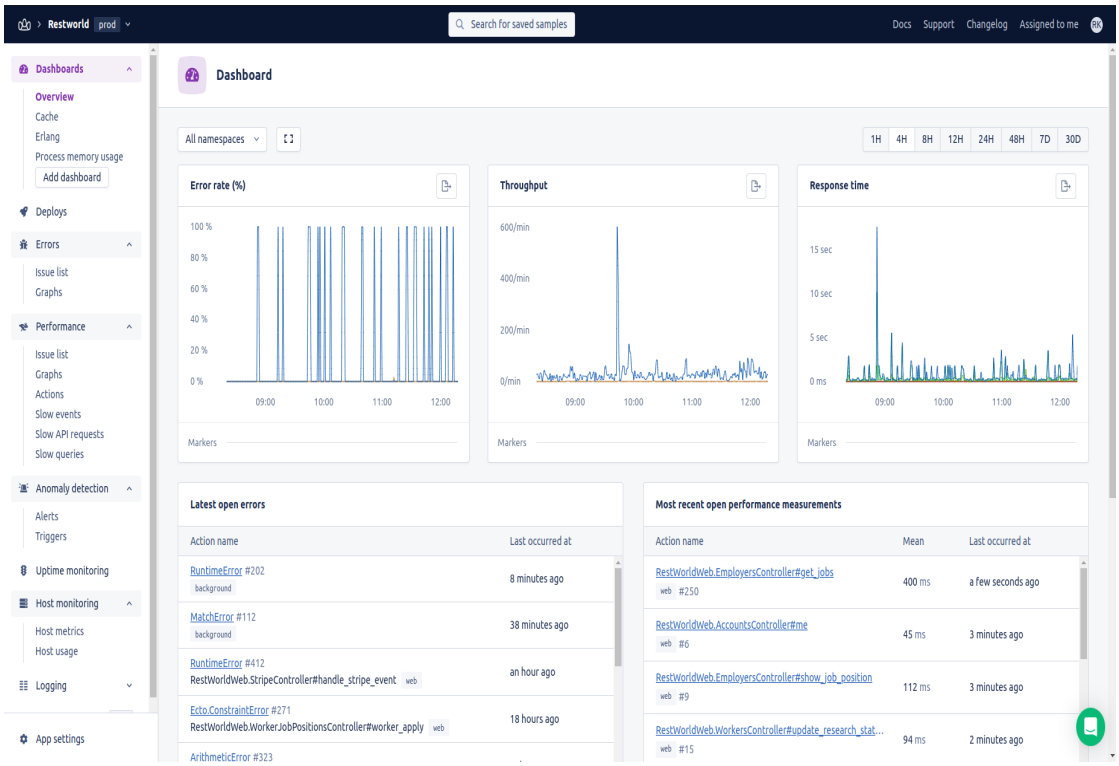

Figure 4.1: Appsignal Dashboard

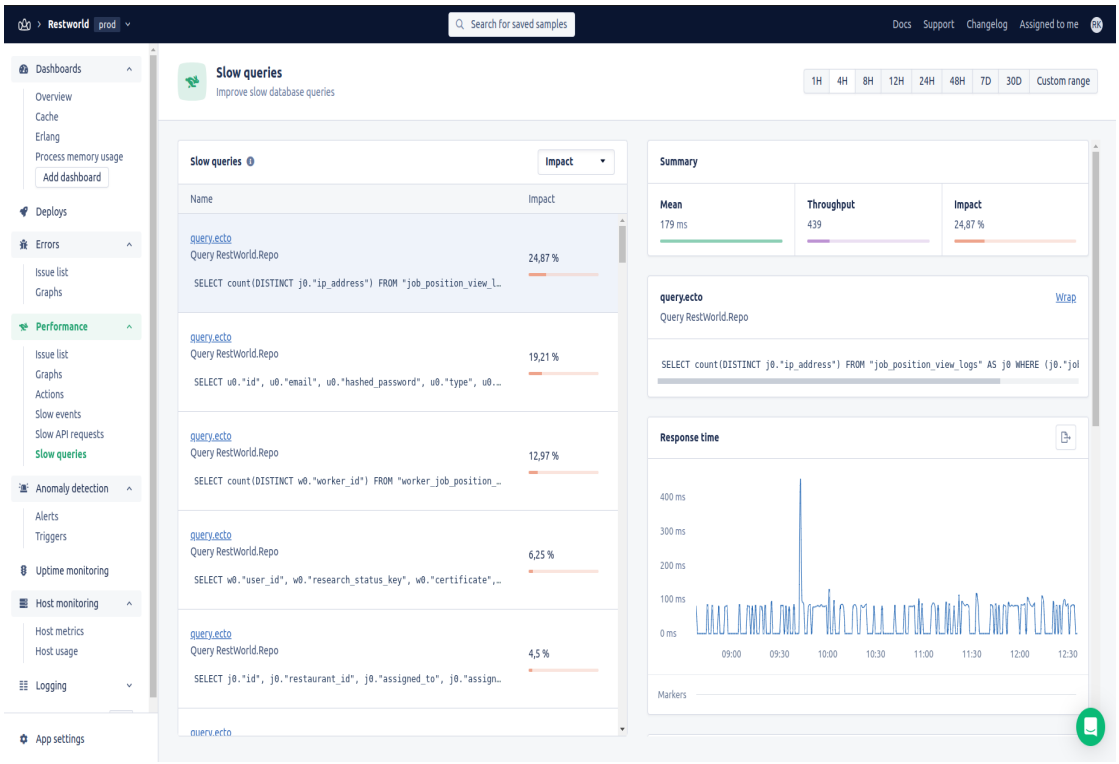

Figure 4.2: Slow Queries Appsignal

Chapter 5

Conclusion

5.0.1 Storing Forms in Database

Initially the registrations forms available in the system were stored in DB as a JSON field. Postgres allows you to define JSON[5] data-type for a column.

Figure 5.1: Forms Table

The **data** field will contain the whole form structure, divided into steps, questions and the possible answers. On front-end, the whole data field is fetched from backend and then rendered based on the data from backend. The problem with this approach is that it's not scalable. When we need to ask more questions in a step, change the question, or the answers, we would need to change it on backend table and also on the client, front-end.

In order to have a scalable system, the approach is not to use the defined forms structure on backend, rather it is defined at the front-end. We have the required data fields defined in each of the schema. The front-end will send the payload containing the data and that data is stored in the database entity. Lets take an example of a worker registration form. In that form, if the worker fills a step which required him to enter the address fields, then the data payload to the backend would be,

```
{
    "user_id": "2SL96rtz4FkwCZLoB0RdIC46HBm",
    "address": {
        "address_lat": "39.8",
        "address_lon": "11.3",
        "city": "Torino",
        "country": "Italia"
    }
}
```

After receiving the payload, the worker with the give user_id will be updated with the address fields present.

5.0.2 Using GraphQl

GraphQl is a query language which allows efficient data-fetching by providing a query of the required data fields. It removes the problem of over-fetching of data [6].

Let's take an example. We have a restaurant entity as shown in figure 5.2. Now there is a case in which we only need the restaurant_name to be shown on page. If we use the REST API to GET the restaurant information using id, it will return all the other information with it which is not required right now, thus consuming network bandwidth. If we implement graphql for this endpoint, we can define the fields that are required directly in the query on front-end. The query can be,

```
query get_restaurant {
    id,
```

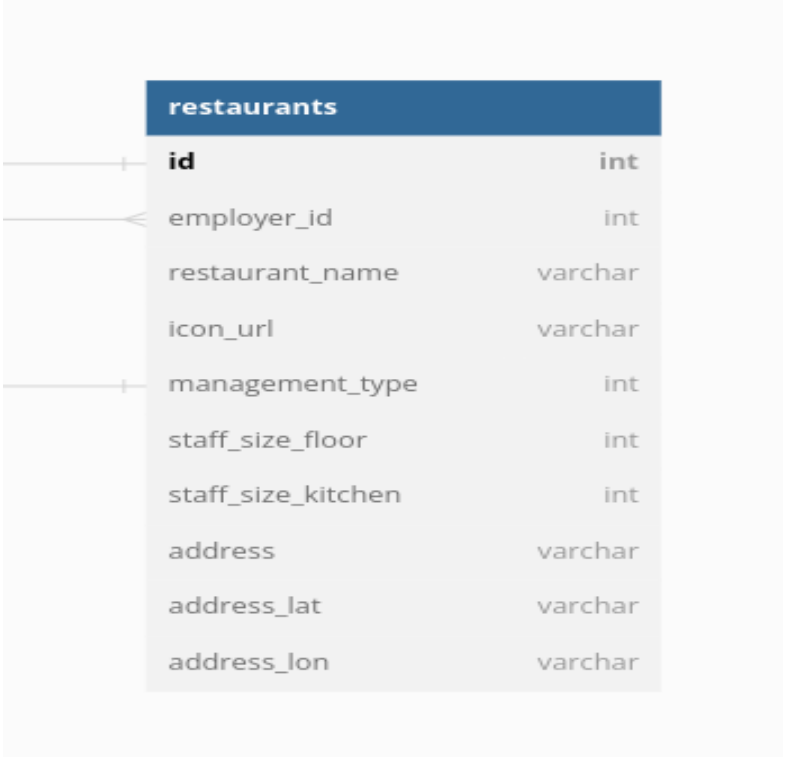

Figure 5.2: Restaurants Table

```
        restaurant_name
    }
```

By defining only the required fields, we can decrease the consumption of network bandwidth. Here, in the following table, is a difference, in terms of speed and size of the payload, between graphql and REST API for an endpoint which fetches a job position.

Table 5.1: GraphQl vs REST

Technology	Payload Size (kB)	Time (ms)
Using REST API	297	1016
Uisng GraphQl	3.7	313

Using graphql endpoints can help improve the overall product performance. By reducing the network bandwidth consumption, we can speed up the data rendering on client.

large font Large font LARGE font

Bibliography

[1] Sonali Panda. *SDLC (Software Development Life Cycle) Phases, Process. What is SDLC*. 2023. URL: https://www.numpyninja.com/post/sdlc-software-development-life-cycle-phases-process-what-is-sdlc (cit. on p. 2).

[2] Anthony Thong Do. *Top 5 Free Database Diagram Design Tools*. 2018. URL: https://www.holistics.io/blog/top-5-free-database-diagram-design-tools/ (cit. on p. 4).

[3] T. Pattinson. *Relational vs Non-Relational Databases*. 2020. URL: https://www.pluralsight.com/blog/software-development/relational-vs-non-relational-databases (cit. on p. 7).

[4] Stanislav Vishnevskiy. *How Discord Scaled Elixir to 5,000,000 Concurrent Users*. 2017. URL: https://discord.com/blog/how-discord-scaled-elixir-to-5-000-000-concurrent-users (cit. on p. 9).

[5] PostgreSQL. *JSON Types*. = https://www.postgresql.org/docs/current/datatype-json.html, (cit. on p. 30).

[6] *GraphQL - A query language for APIs*. https://graphql.org/. Accessed: July 9, 2023 (cit. on p. 31).