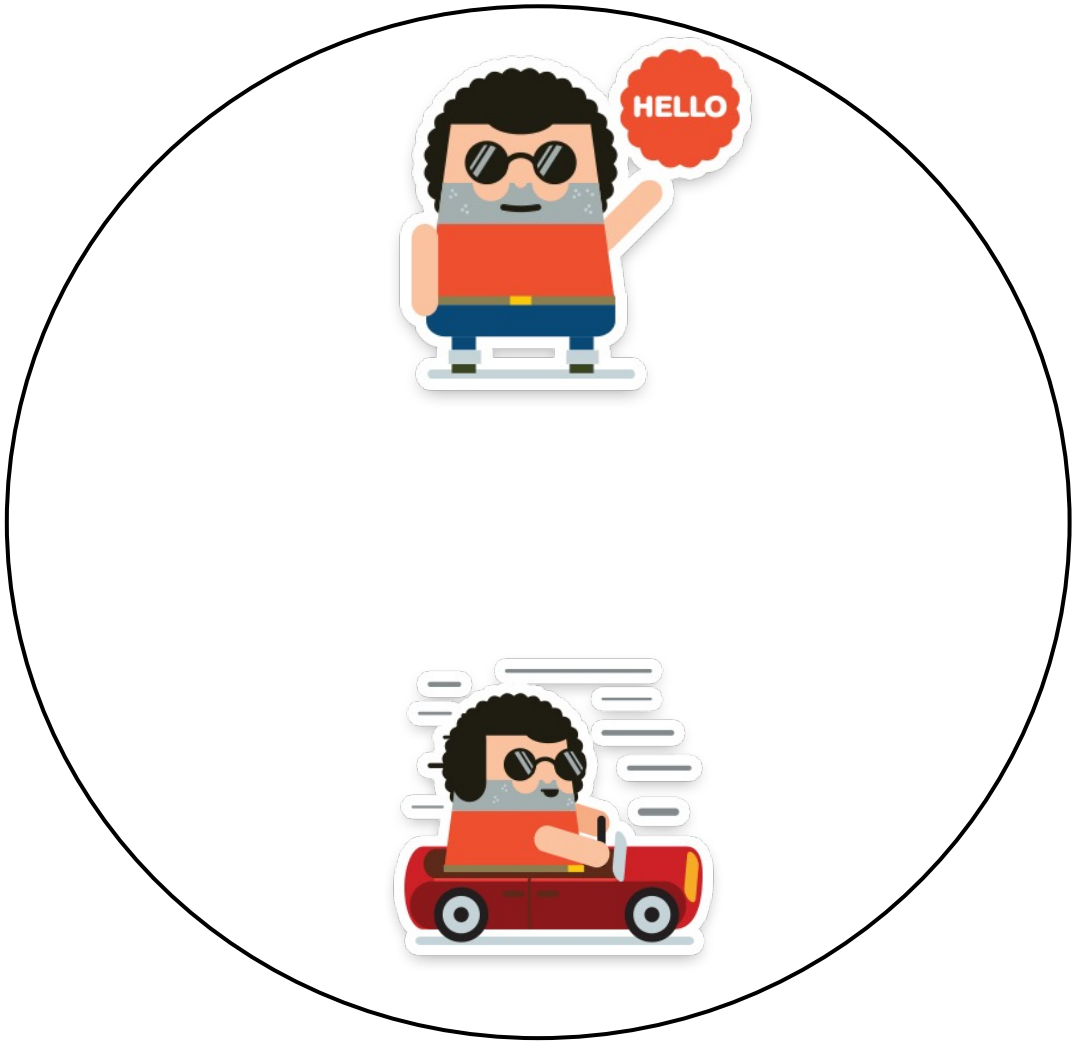


The problem domain, as explained to you by the domain experts

User

Car

User		
Uuid	id	
String	name	
String	surname	
i32	age	
String	email	

Car		
Uuid	id	
Uuid	owned_by	
String	brand	
i32	price	

User		
Uuid	id	
String	name	
String	surname	
i32	age	
String	email	

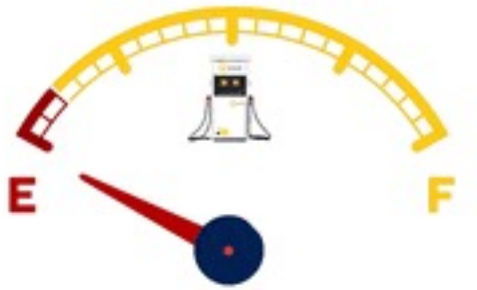
Owns a car

Car		
Uuid	id	
Uuid	owned_by	
String	brand	
i32	price	

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	



User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
MyService-db:
    image: postgres:13-alpine
    container_name: MyService-db
    restart: always
    POSTGRES_USER: "postgres"
    POSTGRES_PASSWORD: "postgres"
    ports:
        - 5432:5432
    volumes:
        - "MyService-data:/var/lib/postgresql/data"
        - "./docker/postgres/01.sql:/docker-entrypoint-
            initdb.d/01_db.sql"
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
MyService-db:
    image: postgres:13-alpine
    container_name: MyService-db
    restart: always
    POSTGRES_USER: "postgres"
    POSTGRES_PASSWORD: "postgres"
    ports:
        - 5432:5432
    volumes:
        - "MyService-data:/v
        - "./docker/postgres
            initdb.d/01_db.s
```

```
DROP DATABASE IF EXISTS "my_service_db";

CREATE DATABASE "my_service_db";
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
MyService-db:
    image: postgres:13-alpine
    container_name: MyService-db
    restart: always
    POSTGRES_USER: "postgres"
    POSTGRES_PASSWORD: "postgres"
    ports:
        - 5432:5432
    volumes:
        - "MyService-data:/v
        - "./docker/postgres
            initdb.d/01_db.s
```

```
DROP DATABASE IF EXISTS "my_service_db";

 DATABASE "my_service_db";
```

```
CREATE TABLE IF NOT EXISTS users (
age INT NOT NULL,
email VARCHAR(255) NOT NULL,
id UUID NOT NULL,
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS cars (
brand VARCHAR(255) NOT NULL,
id UUID NOT NULL,
owned_by UUID NOT NULL
);
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
MyService-db:
    image: postgres:13-alpine
    container_name: MyService-db
    restart: always
    POSTGRES_USER: "postgres"
    POSTGRES_PASSWORD: "postgres"
    ports:
        - 5432:5432
    volumes:
        - "MyService-data:/v
        - "./docker/postgres
            initdb.d/01_db.s
```

```
DROP DATABASE IF EXISTS "my_service_db";

CREATE DATABASE "my_service_db";
```

```
CREATE TABLE IF NOT EXISTS users (
age INT NOT NULL,
email VARCHAR(255) NOT NULL,
id UUID NOT NULL,
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS cars (
brand VARCHAR(255) NOT NULL,
```

```
CREATE INDEX IF NOT EXISTS users_name_index ON users (name);
CREATE INDEX IF NOT EXISTS users_age_index ON users (age);
CREATE INDEX IF NOT EXISTS users_surname_index ON users
(name,surname);
CREATE INDEX IF NOT EXISTS users_email_index ON users (email);
CREATE UNIQUE INDEX IF NOT EXISTS unique_email_index ON users
(email);
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
MyService-db:
    image: postgres:13-alpine
    container_name: MyServi
    restart: always
    POSTGRES_USER: "postgre
    POSTGRES_PASSWORD: "pos
    ports:
        - 5432:5432
    volumes:
```

```
pub struct User {
    pub age: i32,
    pub email: String,
    pub id: Uuid,
    pub name: String,
    pub surname: String,
}
```

```
pub struct Car {
    pub brand: String,
    pub id: Uuid,
    pub owned_by: Uuid,
}
```

```
DROP DATABASE IF EXISTS "my_service_db";

 DATABASE "my_service_db";
```

```
CREATE
age INT
email VARCHAR(255) NOT NULL,
id UUID NOT NULL,
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS cars (
brand VARCHAR(255) NOT NULL,
```

```
CREATE INDEX IF NOT EXISTS users_name_index ON users (name);
CREATE INDEX IF NOT EXISTS users_age_index ON users (age);
CREATE INDEX IF NOT EXISTS users_surname_index ON users
(name,surname);
CREATE INDEX IF NOT EXISTS users_email_index ON users (email);
CREATE UNIQUE INDEX IF NOT EXISTS unique_email_index ON users
(email);
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
Router::new()
        .route("/v1/users", get(get_users).post(create_user))
        .route("/v1/users/:id", get(get_user).put(update_user).delete(delete_user))
        .route("/v1/cars", get(get_cars).post(create_car))
        .route("/v1/cars/:id", get(get_car).put(update_car).delete(delete_car))
        .with_state(services)
```

```
        - 5432:5432
    volumes:
```

```
pub struct Car {
    pub brand: String,
    pub id: Uuid,
    pub owned_by: Uuid,
}
```

```
    pub name: String,
    pub surname: String,
}
```

```
DROP DATABASE IF EXISTS "my_service_db";

CREATE DATABASE "my_service_db";
```

```
CREATE
age INT
email VARCHAR(255) NOT NULL,
id UUID NOT NULL,
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL
);
```

```
CREATE TABLE IF NOT EXISTS cars (
brand VARCHAR(255) NOT NULL,
```

```
CREATE INDEX IF NOT EXISTS users_name_index ON users (name);
CREATE INDEX IF NOT EXISTS users_age_index ON users (age);
CREATE INDEX IF NOT EXISTS users_surname_index ON users
(name,surname);
CREATE INDEX IF NOT EXISTS users_email_index ON users (email);
CREATE UNIQUE INDEX IF NOT EXISTS unique_email_index ON users
(email);
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
Router::new()
        .route("/v1/users", get(get_users).post(create_user))
        .route("/v1/users/:id", get(get_user).put(update_user).delete(delete_user))
        .route("/v1/cars", get(get_cars).post(create_car))
        .route("/v1/cars/:id", get(get_car).put(update_car).delete(delete_car))
        .with_state(services)
```

```
        - 5432:5432
    volumes:
```

```
    pub name: String,
    pub surname: String,
}
```

```
pub struct Car {
    pub brand: String,
    pub id: Uuid,
}
```

```
DROP DATABASE IF EXISTS "my_service_db";

CREATE DATABASE "my_service_db";
```

```
CREATE
age INT
email VARC
id UUID NO
name VARCHAR(255) NOT NULL,
surname VARCHAR(255) NOT NULL
);
```

```
match self.get_car(&car.id).await {
    Ok(_) => (),
    Err(e) => return Err(Error::CarDoesNotExist)
};
```

```
CREATE TABLE IF NOT EXISTS cars (
brand VARCHAR(255) NOT NULL,
```

```
CREATE INDEX IF NOT EXISTS users_name_index ON users (name);
CREATE INDEX IF NOT EXISTS users_age_index ON users (age);
CREATE INDEX IF NOT EXISTS users_surname_index ON users
(name,surname);
CREATE INDEX IF NOT EXISTS users_email_index ON users (email);
CREATE UNIQUE INDEX IF NOT EXISTS unique_email_index ON users
(email);
```

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
Router::new()
        .route("/v1/users", get(get_users).post(creat
```

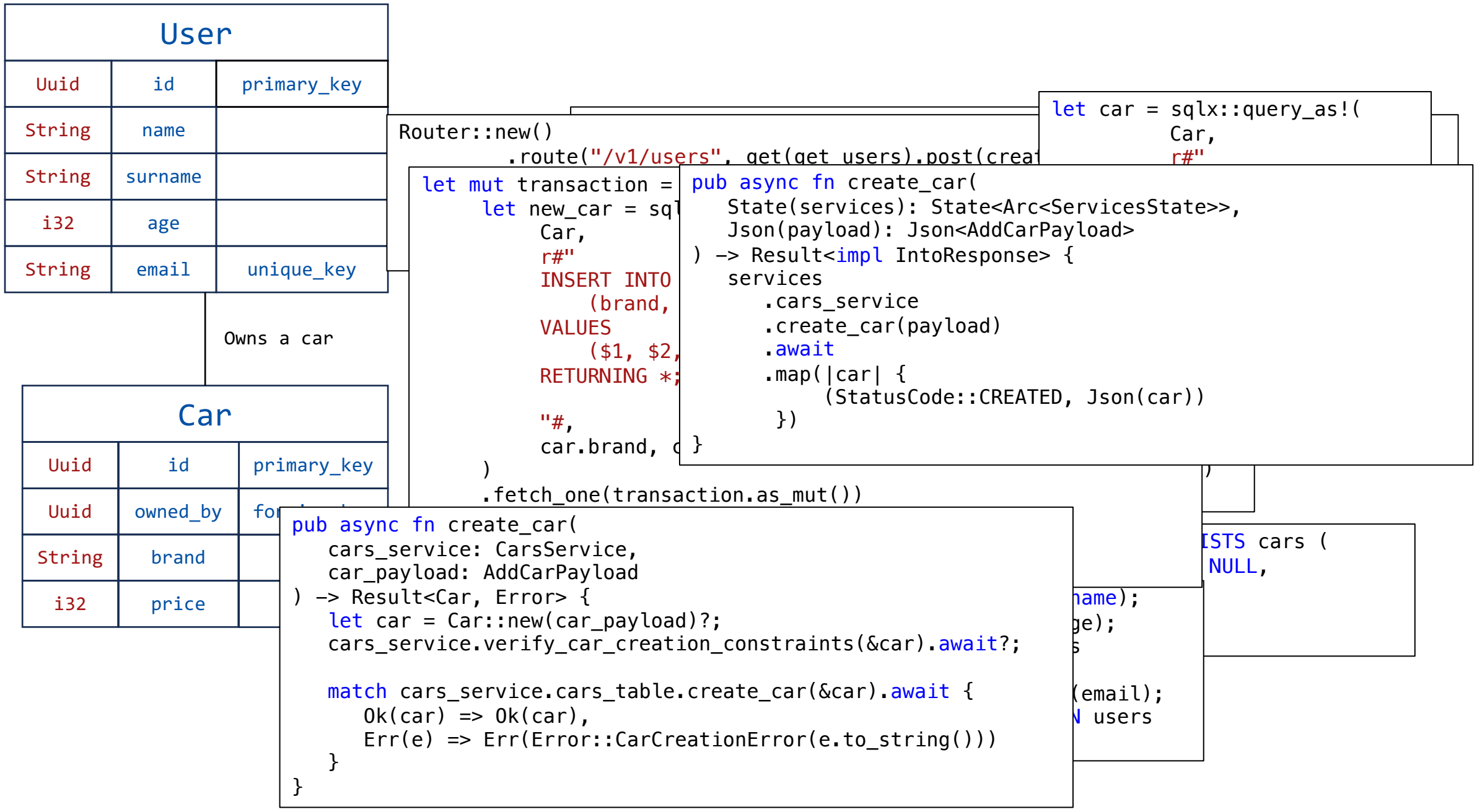
```
let mut transaction = self.pool.begin().await?;
     let new_car = sqlx::query_as!(
          Car,
          r#"
          INSERT INTO cars
              (brand, id, name, owned_by)
          VALUES
              ($1, $2, $3, $4)
          RETURNING *;

          "#,
          car.brand, car.id, car.name, car.owned_by,
     )
     .fetch_one(transaction.as_mut())
     .await?;
    transaction.commit().await?;
    Ok(new_car)
```

```
let car = sqlx::query_as!(
          Car,
          r#"
          SELECT * FROM cars
          WHERE id = $1;
          "#,
          id
)
.fetch_one(self.pool.as_ref())
.await?;
Ok(car)
```

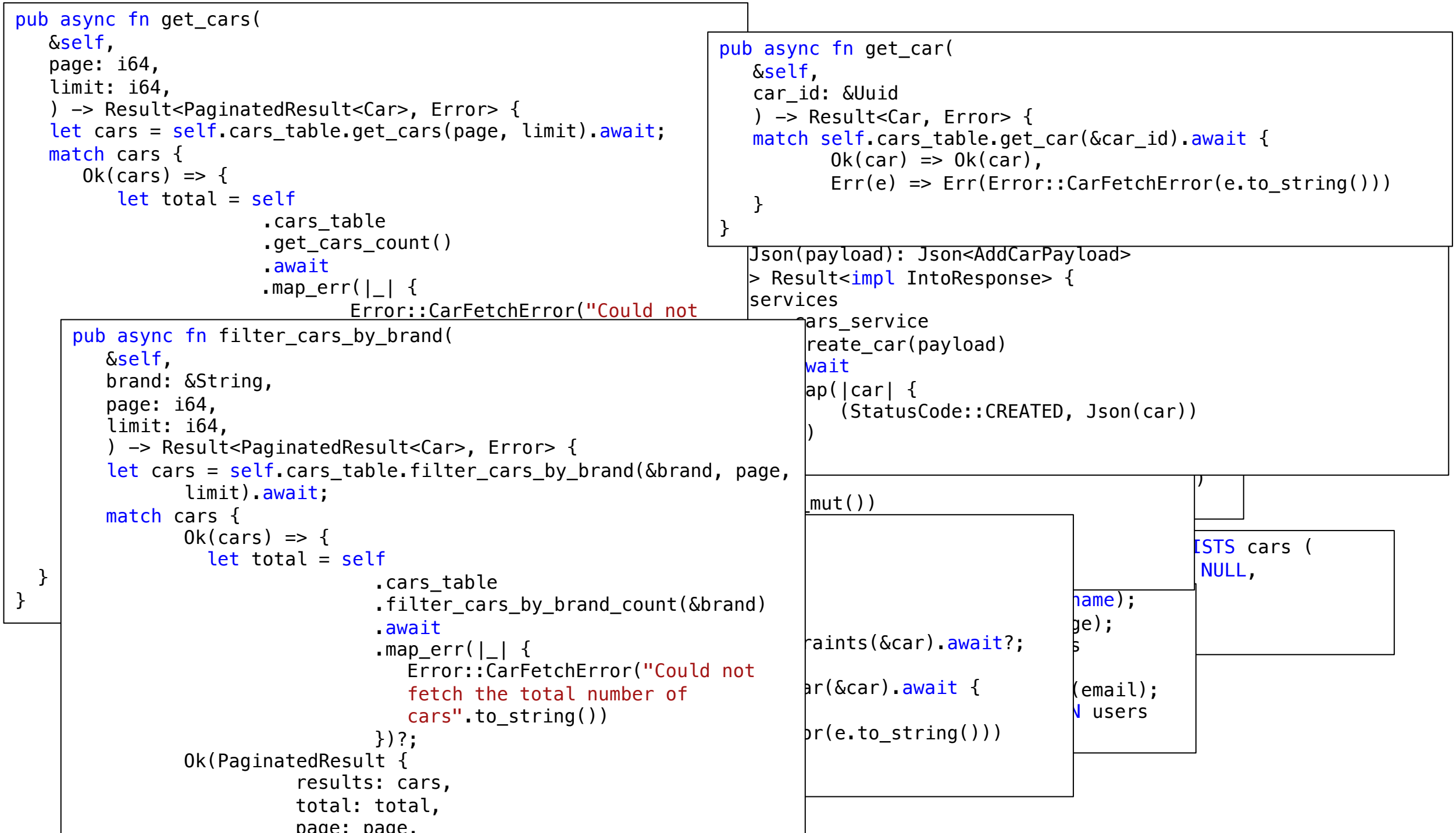
```
_db";
)
```

```
ISTS cars (
 NULL,
```

```
CREATE INDEX IF NOT EXISTS users_name_index ON users (name);
CREATE INDEX IF NOT EXISTS users_age_index ON users (age);
CREATE INDEX IF NOT EXISTS users_surname_index ON users
(name,surname);
CREATE INDEX IF NOT EXISTS users_email_index ON users (email);
CREATE UNIQUE INDEX IF NOT EXISTS unique_email_index ON users
(email);
```


User
Uuid
id
primary_key
String
name
String
surname
i32
age
String
email
unique_key
Owns a car
Car
Uuid
id
primary_key
Uuid
owned_by
String
brand
i32
price
Router::new()
.route("/v1/users", get(get_users).post(creat
let mut transaction =
let new_car = sql
Car,
r#"
INSERT INTO
(brand,
VALUES
($1, $2,
RETURNING *;
"#,
car.brand, c
)
.fetch_one(transaction.as_mut())
let car = sqlx::query_as!(
Car,
r#"
pub async fn create_car(
State(services): State<Arc<ServicesState>>,
Json(payload): Json<AddCarPayload>
) -> Result<impl IntoResponse> {
services
.cars_service
.create_car(payload)
.await
.map(|car| {
(StatusCode::CREATED, Json(car))
})
}
pub async fn create_car(
cars_service: CarsService,
car_payload: AddCarPayload
) -> Result<Car, Error> {
let car = Car::new(car_payload)?;
cars_service.verify_car_creation_constraints(&car).await?;
match cars_service.cars_table.create_car(&car).await {
Ok(car) => Ok(car),
Err(e) => Err(Error::CarCreationError(e.to_string()))
}
}
ISTS cars (
NULL,
name);
ge);
s
(email);
N users


pub async fn get_cars(
&self,
page: i64,
limit: i64,
) -> Result<PaginatedResult<Car>, Error> {
let cars = self.cars_table.get_cars(page, limit).await;
match cars {
Ok(cars) => {
let total = self
.cars_table
.get_cars_count()
.await
.map_err(|_| {
Error::CarFetchError("Could not
fetch the total number of
cars".to_string())
})?;
Ok(PaginatedResult {
results: cars,
total: total,
page: page,
page_size: limit,
})
}
Err(e) => Err(Error::CarFetchError(e.to_string())),
}
}
pub async fn get_car(
&self,
car_id: &Uuid
) -> Result<Car, Error> {
match self.cars_table.get_car(&car_id).await {
Ok(car) => Ok(car),
Err(e) => Err(Error::CarFetchError(e.to_string()))
}
}
Json(payload): Json<AddCarPayload>
> Result<impl IntoResponse> {
services
.cars_service
.create_car(payload)
.await
.map(|car| {
(StatusCode::CREATED, Json(car))
})
on.as_mut())
ISTS cars (
NULL,
name);
ge);
s
(email);
N users
cars_service.verify_car_creation_constraints(&car).await?;
match cars_service.cars_table.create_car(&car).await {
Ok(car) => Ok(car),
Err(e) => Err(Error::CarCreationError(e.to_string()))
}
}


pub async fn get_cars(
&self,
page: i64,
limit: i64,
) -> Result<PaginatedResult<Car>, Error> {
let cars = self.cars_table.get_cars(page, limit).await;
match cars {
Ok(cars) => {
let total = self
.cars_table
.get_cars_count()
.await
.map_err(|_| {
Error::CarFetchError("Could not
pub async fn get_car(
&self,
car_id: &Uuid
) -> Result<Car, Error> {
match self.cars_table.get_car(&car_id).await {
Ok(car) => Ok(car),
Err(e) => Err(Error::CarFetchError(e.to_string()))
}
}
Json(payload): Json<AddCarPayload>
> Result<impl IntoResponse> {
services
ars_service
reate_car(payload)
wait
ap(|car| {
(StatusCode::CREATED, Json(car))
)
mut())
ISTS cars (
NULL,
name);
ge);
s
(email);
N users
raints(&car).await?;
ar(&car).await {
or(e.to_string()))
pub async fn filter_cars_by_brand(
&self,
brand: &String,
page: i64,
limit: i64,
) -> Result<PaginatedResult<Car>, Error> {
let cars = self.cars_table.filter_cars_by_brand(&brand, page,
limit).await;
match cars {
Ok(cars) => {
let total = self
.cars_table
.filter_cars_by_brand_count(&brand)
.await
.map_err(|_| {
Error::CarFetchError("Could not
fetch the total number of
cars".to_string())
})?;
Ok(PaginatedResult {
results: cars,
total: total,
page: page,
}
}

```rust
pub async fn get_cars(
   &self,
   page: i64,
   limit: i64,
   ) -> Result<PaginatedResult<Car>, Error> {
   let cars = self.cars_table.get_cars(page, limit).await;
   match cars {
      Ok(cars) => {
         let total = self
                     .cars_table
                     .get_cars_count()
                     .await
                     .map_err(|_| {
                              Error::CarFetchError("Could not
```

```rust
pub async fn get_car(
   &self,
   car_id: &Uuid
   ) -> Result<Car, Error> {
   match self.cars_table.get_car(&car_id).await {
         Ok(car) => Ok(car),
         Err(e) => Err(Error::CarFetchError(e.to_string()))
   }
}
```

```rust
Json(payload): Json<AddCarPayload>
> Result<impl IntoResponse> {
services
   .cars_service
   .create_car(payload)
   .await
   .map(|car| {
      (StatusCode::CREATED, Json(car))
   )
```

```rust
pub async fn filter_cars_by_brand(
   &self,
   brand: &String,
   page: i64,
   limit: i64,
   ) -> Result<PaginatedResult<Car>, Error> {
   let cars = self.cars_table.filter_cars_by_brand(&brand, p
         limit).await;
   match cars {
         Ok(cars) => {
           let total = self
                        .cars_table
                        .filter_cars_by_brand_count(&bran
                        .await
                        .map_err(|_| {
                           Error::CarFetchError("Could no
                           fetch the total number of
                           cars".to_string())
                        })?;
         Ok(PaginatedResult {
                  results: cars,
                  total: total,
                  page: page,
```

```rust
pub async fn update_car(
   &self,
   car_id: &Uuid,
   car_payload: UpdateCarPayload
   ) -> Result<Car, Error> {
    let car = self.get_car(car_id).await?.update(car_payload)?;
    self.verify_car_update_constraints(&car).await?;

    match self.cars_table.update_car(&car).await {
         Ok(car) => Ok(car),
         Err(e) => Err(Error::CarUpdateError(e.to_string()))
   }
}
```

```rust
pub async fn get_cars(
   &self,
   page: i64,
   limit: i64,
   ) -> Result<PaginatedResult<Car>, Error> {
   let cars = self.cars_table.get_cars(page, limit).await;
   match cars {
      Ok(cars) => {
         let total = self
                  .
                  .
                  .
                  .n
```

```rust
pub async fn get_car(
   &self,
   car_id: &Uuid
   ) -> Result<Car, Error> {
   match self.cars_table.get_car(&car_id).await {
         Ok(car) => Ok(car),
         Err(e) => Err(Error::CarFetchError(e.to_string()))
   }
```

```rust
oad>
```

```rust
Json(car))
```

```rust
pub async fn filte
   &self,
   brand: &String,
   page: i64,
   limit: i64,
   ) -> Result<Pag
   let cars = self
         limit).a
   match cars {
         Ok(cars)
           let to
  }
}
```

```rust
oad
```

```rust
r_id).await?.update(car_payload)?;
self.verify_car_update_constraints(&car).await?;

match self.cars_table.update_car(&car).await {
      Ok(car) => Ok(car),
      Err(e) => Err(Error::CarUpdateError(e.to_string()))
   }
}
```

```rust
                  .await
                  .map_err(|_| {
                     Error::CarFetchError("Could no
                     fetch the total number of
                     cars".to_string())
                  })?;
         Ok(PaginatedResult {
                  results: cars,
                  total: total,
                  page: page,
```

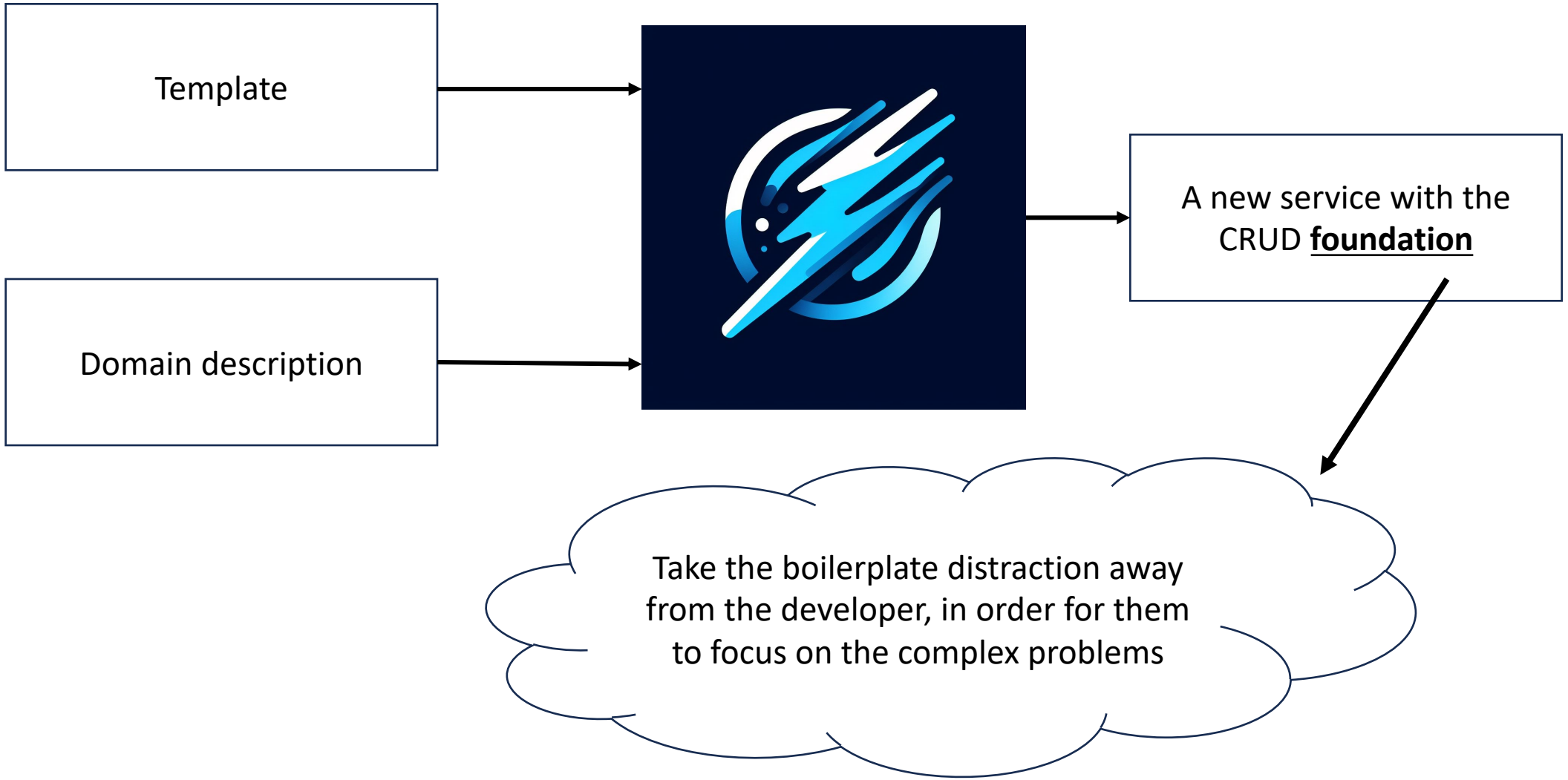
```rust
pub async fn get_cars(
   &self,
   page: i64,
   limit: i64,
   ) -> Result<PaginatedResult<Car>, Error> {
   let cars = self.cars_table.get_cars(page, limit).await;
   match cars {
      Ok(cars) => {
         let total = self
                     .cars_table
                     .get_cars_count()
                     .await
                     .map_err(|_| {
                              Error::CarFetchError("Could not
```

```rust
pub async fn get_car(
   &self,
   car_id: &Uuid
   ) -> Result<Car, Error> {
   match self.cars_table.get_car(&car_id).await {
         Ok(car) => Ok(car),
         Err(e) => Err(Error::CarFetchError(e.to_string()))
   }
}
```

```rust
Json(payload): Json<AddCarPayload>
> Result<impl IntoResponse> {
services
   .cars_service
   .create_car(payload)
   .await
   .map(|car| {
      (StatusCode::CREATED, Json(car))
   )
```

```rust
pub async fn filter_cars_by_brand(
   &self,
   brand: &String,
   page: i64,
   limit: i64,
   ) -> Result<PaginatedResult<Car>, Error> {
   let cars = self.cars_table.filter_cars_by_brand(&brand, p
         limit).await;
   match cars {
         Ok(cars) => {
           let total = self
                        .cars_table
                        .filter_cars_by_brand_count(&bran
                        .await
                        .map_err(|_| {
                           Error::CarFetchError("Could no
                           fetch the total number of
                           cars".to_string())
                        })?;
         Ok(PaginatedResult {
                  results: cars,
                  total: total,
                  page: page,
```

```rust
pub async fn update_car(
   &self,
   car_id: &Uuid,
   car_payload: UpdateCarPayload
   ) -> Result<Car, Error> {
    let car = self.get_car(car_id).await?.update(car_payload)?;
    self.verify_car_update_constraints(&car).await?;

    match self.cars_table.update_car(&car).await {
         Ok(car) => Ok(car),
         Err(e) => Err(Error::CarUpdateError(e.to_string()))
   }
}
```


Template
Domain description
A new service with the CRUD **foundation**
Take the boilerplate distraction away from the developer, in order for them to focus on the complex problems

User		
Uuid	id	primary_key
String	name	
String	surname	
i32	age	
String	email	unique_key

Owns a car

Car		
Uuid	id	primary_key
Uuid	owned_by	foreign_key
String	brand	
i32	price	

```
{
    "service_name": "MyService",
    "entities": [
        {
            "User" :
            {
                "id": "Uuid",
                "name" : "String",
                "surname": "String",
                "age": "i32",
                "email": "String",
                "primary_key": "id",
                "filter_by": ["name", "age", ["name", "surname"], "email"],
                "unique_keys": ["email"]
            }
        },
        {
            "Car" :
            {
                "id": "Uuid",
                "brand": "String",
                "price": “i32",
                "owned_by": "User.id",
                "primary_key": "id",
                "filter_by": ["name", "brand"]
            }
        }
    ]
}
```

```
pub struct {entity_name} {
   {entity_attributes}
}

entity_attribute := `pub {attribute_name}: {attribute_type}`
```

```
{
    "service_name": "MyService",
    "entities": [
        {
            "User" :
            {
                "id": "Uuid",
                "name" : "String",
                "surname": "String",
                "age": "i32",
                "email": "String",
                "primary_key": "id",
                "filter_by": ["name", "age", ["name", "surname"], "email"],
                "unique_keys": ["email"]
            }
        },
        {
            "Car" :
            {
                "id": "Uuid",
                "brand": "String",
                "price": “i32",
                "owned_by": "User.id",
                "primary_key": "id",
                "filter_by": ["name", "brand"]
            }
        }
    ]
}
```

```rust
pub struct User {
    pub id: Uuid,
    pub name: String,
    pub surname: String,
    pub age: i32,
    pub email: String,
}

pub struct Car {
    pub id: Uuid,
    pub brand: String,
    pub price: i32,
    pub owned_by: Uuid,
}
```

```
{service_name}-db:
   image: postgres:13-alpine
   container_name: {service_name}-db
   restart: always
   environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
   ports:
      - 5432:5432
volumes:
      - "{service_name}-data:/var/lib/postgresql/data"
      - "./docker/postgres/01.sql:/docker-entrypoint-initdb.d/01_db.sql"
```

```
DROP DATABASE IF EXISTS "{service_name}_db";

CREATE DATABASE "{service_name}_db";
```

```
{
    "service_name": "MyService",
    "entities": [
        {
            "User" :
            {
                "id": "Uuid",
                "name" : "String",
                "surname": "String",
                "age": "i32",
                "email": "String",
                "primary_key": "id",
                "filter_by": ["name", "age", ["name", "surname"], "email"],
                "unique_keys": ["email"]
            }
        },
        {
            "Car" :
            {
                "id": "Uuid",
                "name" : "String",
                "brand": "String",
                "price": “i32",
                "owned_by": "User.id",
                "primary_key": "id",
                "filter_by": ["name", "brand"]
            }
        }
    ]
}
```

```yaml
MyService-db:
   image: postgres:13-alpine
   container_name: MyService-db
   restart: always
   environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
   ports:
      - 5432:5432
volumes:
      - "MyService-data:/var/lib/postgresql/data"
      - "./docker/postgres/01.sql:/docker-entrypoint-initdb.d/01_db.sql"
```

```sql
DROP DATABASE IF EXISTS "MyService_db";

CREATE DATABASE "MyService_db";
```

```
INSERT INTO {plural_entity_name}
    ({attribute_names})
VALUES
    ($1, $2, $3, $4, $5)
RETURNING *;
```

```
SELECT * FROM {table_name}
WHERE {primary_key} = {primary_key_val};
```

```
{
    "service_name": "MyService",
    "entities": [
        {
            "User" :
            {
                "id": "Uuid",
                "name" : "String",
                "surname": "String",
                "age": "i32",
                "email": "String",
                "primary_key": "id",
                "filter_by": ["name", "age", ["name", "surname"], "email"],
                "unique_keys": ["email"]
            }
        },
        {
            "Car" :
            {
                "id": "Uuid",
                "name" : "String",
                "brand": "String",
                "price": “i32",
                "owned_by": "User.id",
                "primary_key": "id",
                "filter_by": ["name", "brand"]
            }
        }
    ]
}
```

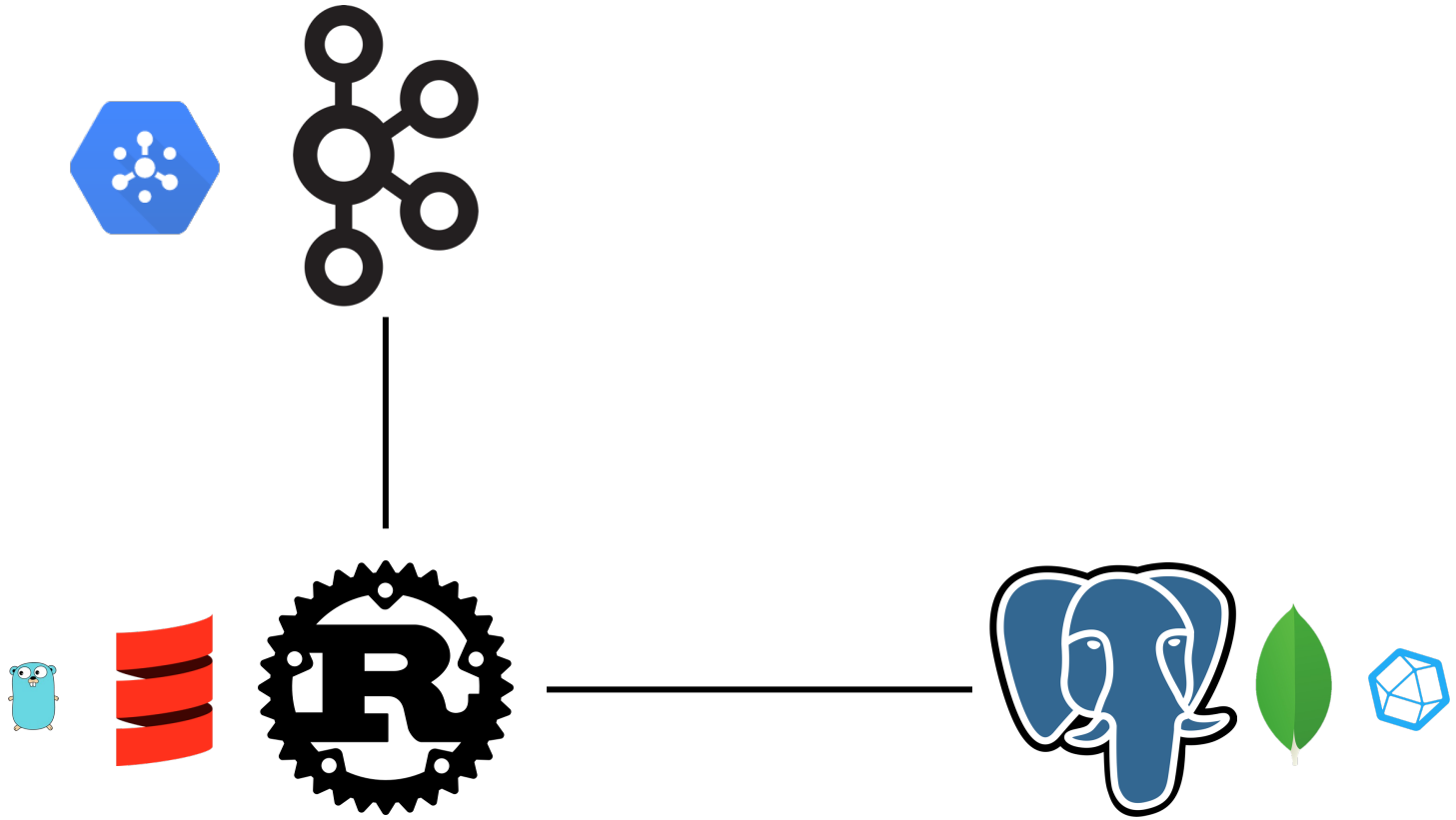
```rust
pub async fn create_car(&self, car: Car) -> Result<Car, Error> {
    let mut transaction = self.pool.begin().await?;
    let new_car= sqlx::query_as!(
                        Car,
                        r#"
                        INSERT INTO cars
                            (brand, id, name, price, owned_by)
                        VALUES
                           ($1, $2, $3, $4, $5)
                        RETURNING *;
                        "#,
                        car.brand, car.id, car.name, car.owned_by,
            )
            .fetch_one(transaction.as_mut())
            .await?;
            transaction.commit().await?;
            Ok(new_car)
}
```

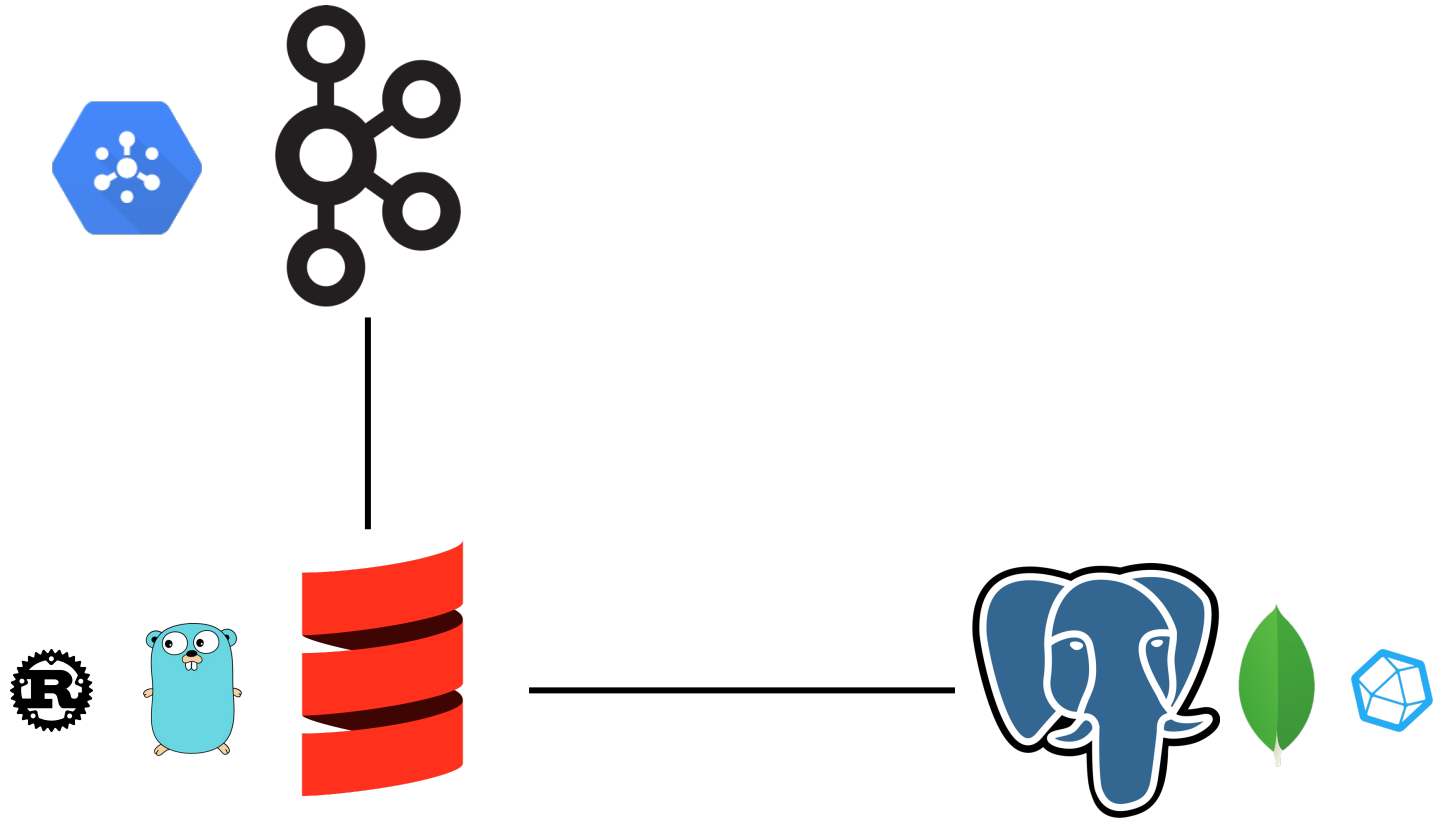
```rust
pub async fn get_car(&self, id: &Uuid) -> Result<Car, Error> {
    let car = sqlx::query_as!(
                        Car,
                       r#"
                       SELECT * FROM cars
                       WHERE id = $1;
                "#,
                 id)
                    .fetch_one(self.pool.as_ref())
                    .await?;
    Ok(car)
}
```

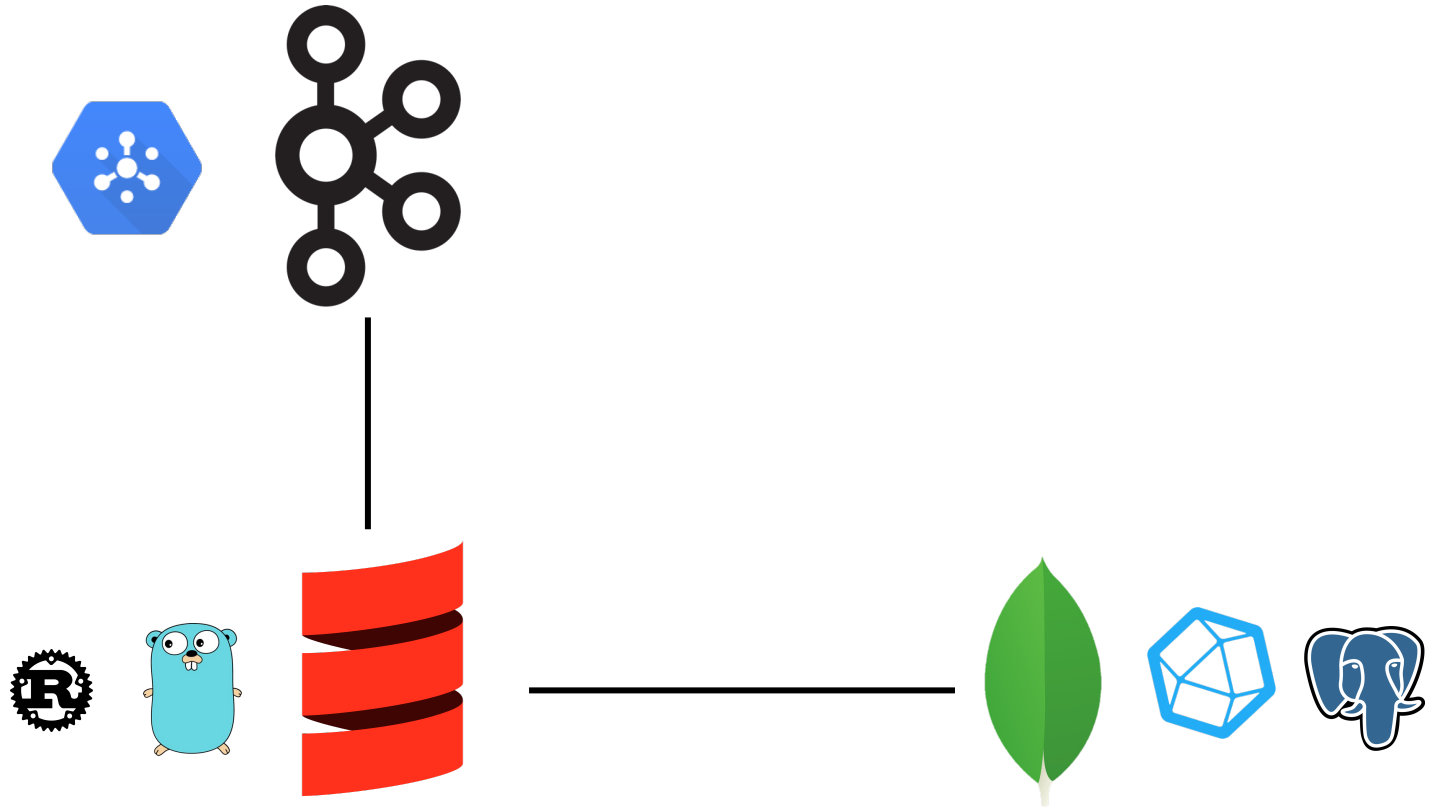
```rust
pub async fn create_car(
   &self,
   Json(payload): Json<AddCarPayload>
) -> Result<impl IntoResponse> {
   let car = Car::new(payload);
    match self.cars_table.create_car(&car).await {
            Ok(car) => Ok((StatusCode::CREATED, Json(car))),
            Err(e) => Err(Error::CarCreationError(e.to_string()))
   }
}
```

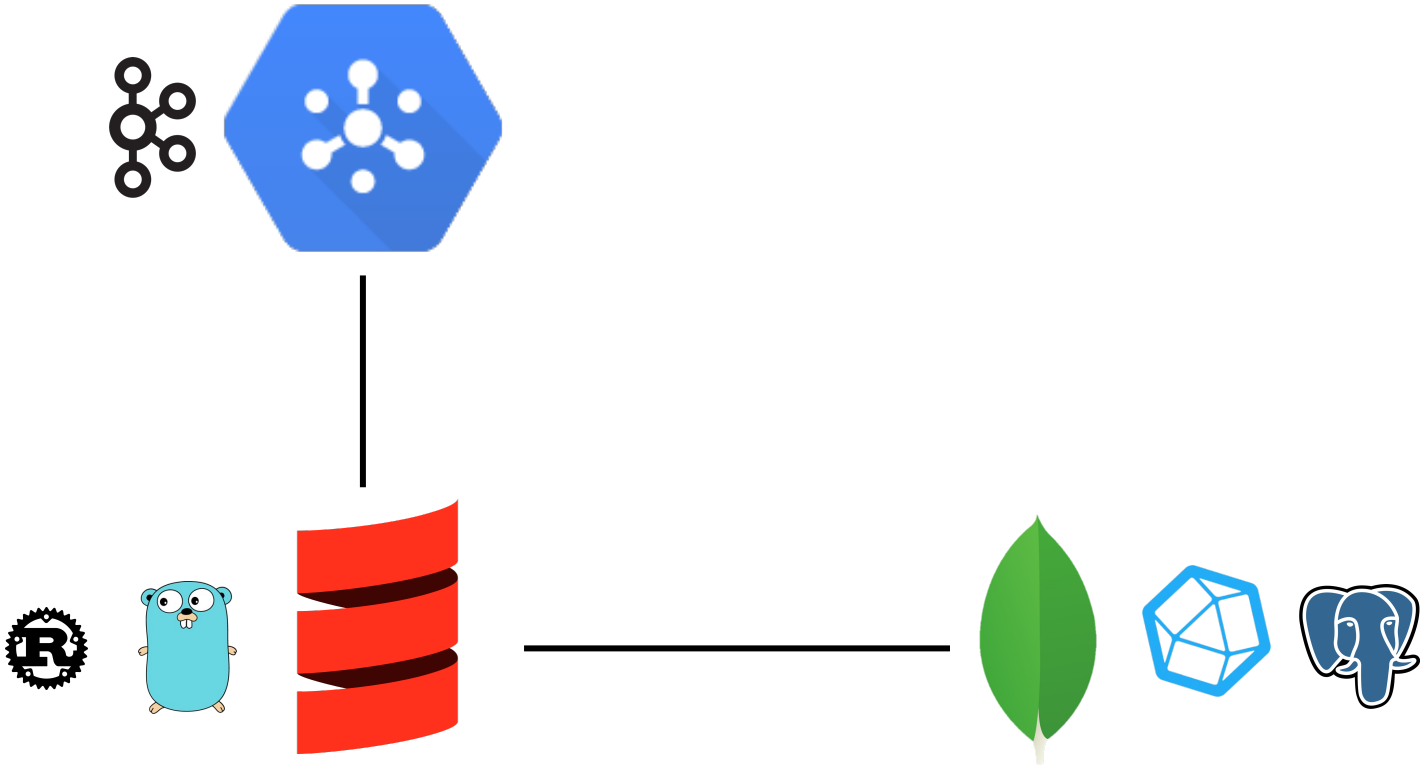
```rust
pub async fn get_car(
   &self,
   Path(id): Path<Uuid>
   ) -> Result<impl IntoResponse> {
   match self.cars_table.get_car(&id).await {
            Ok(car) => Ok(Json(car)),
            Err(e) => Err(Error::CarFetchError(e.to_string()))
   }
}
```

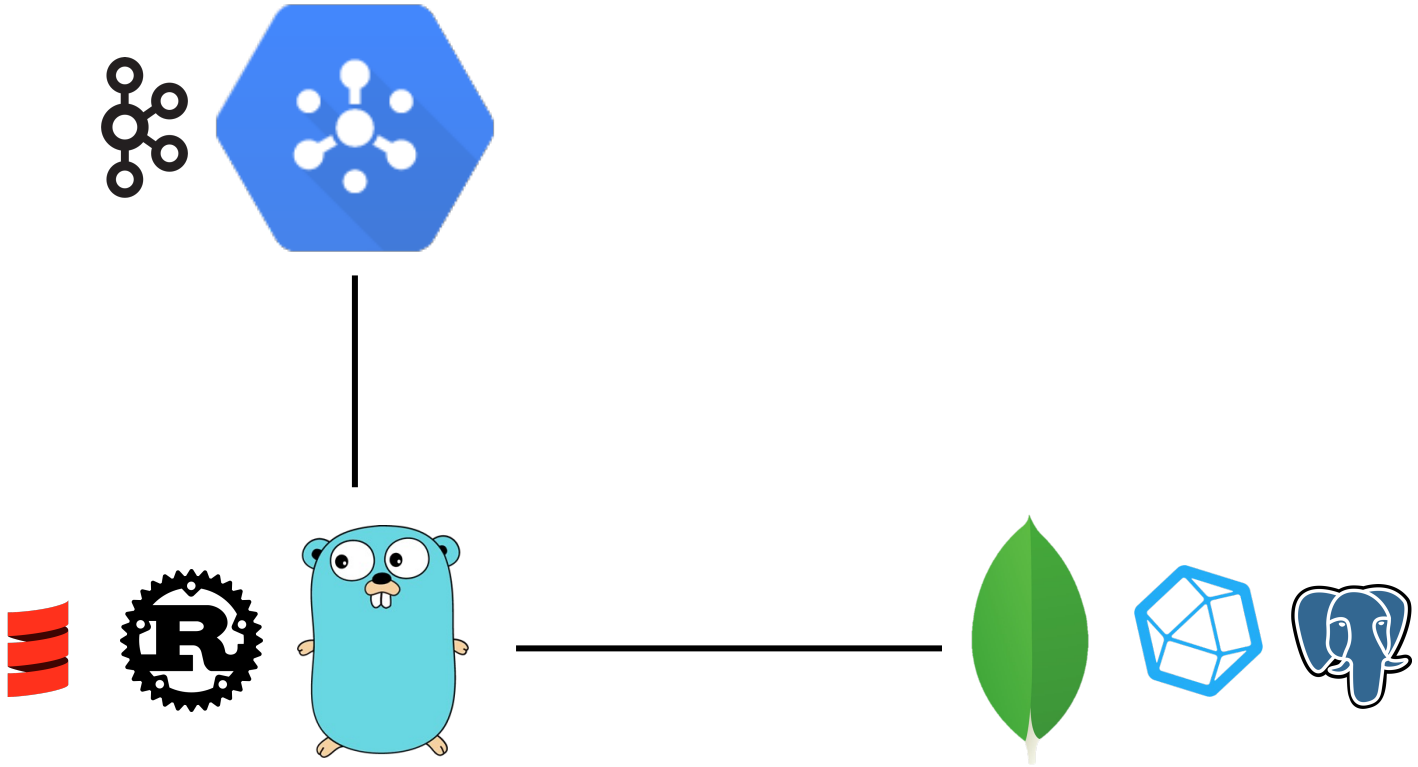
```
Router::new()
          …
         .route("/v1/cars", post(create_car))
         .route("/v1/cars/:id", get(get_car))
         .with_state(services)
```

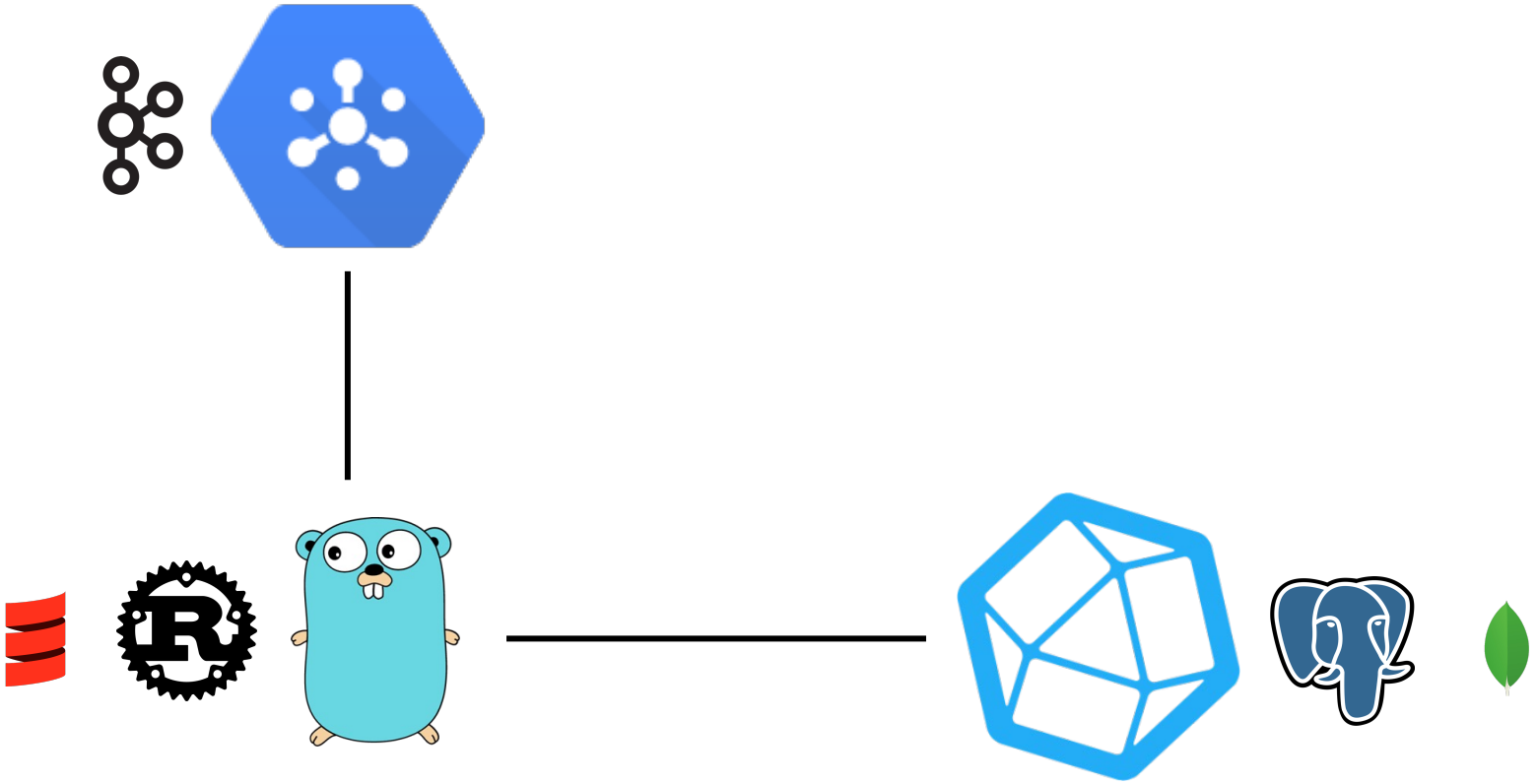


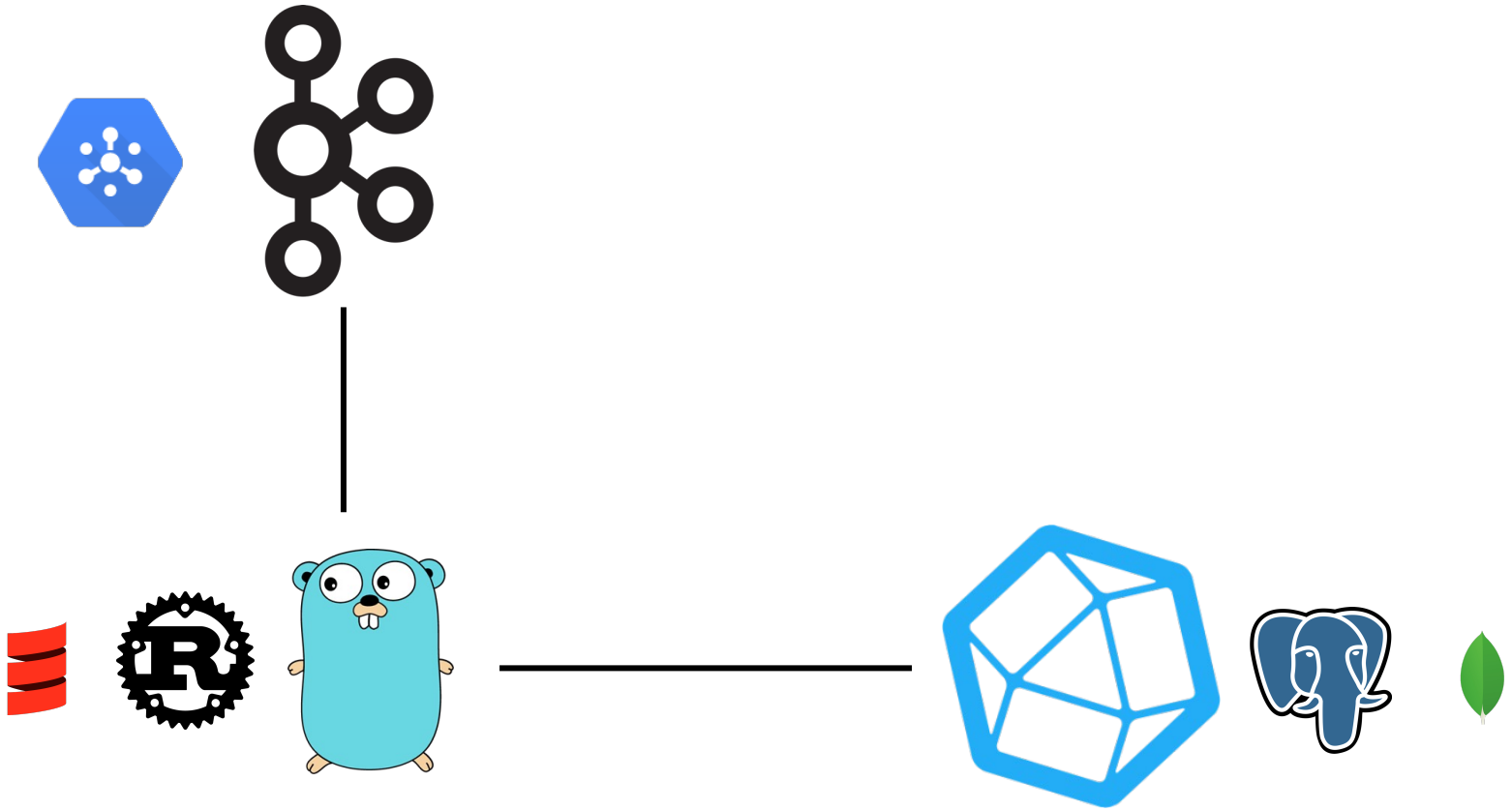


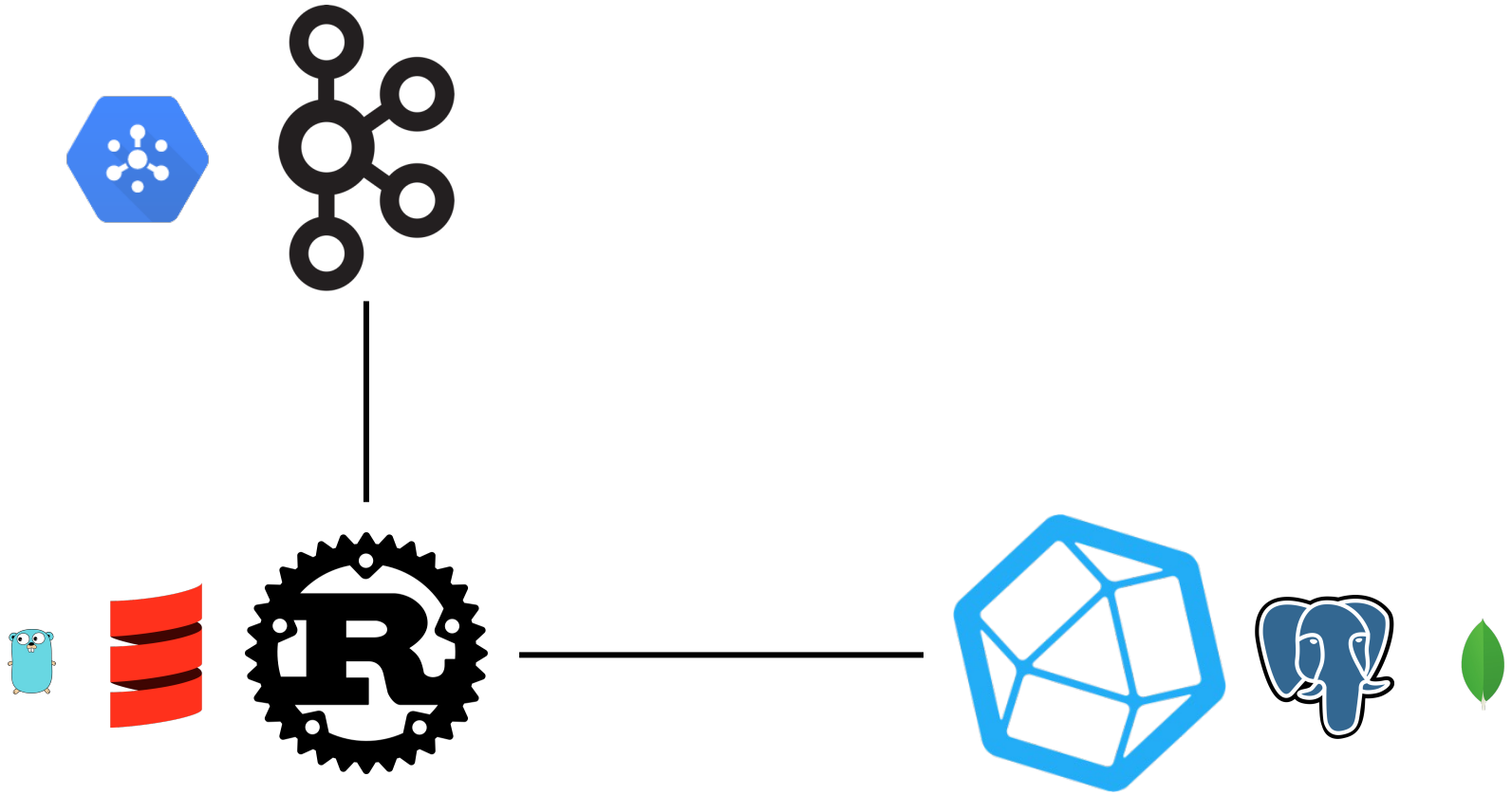


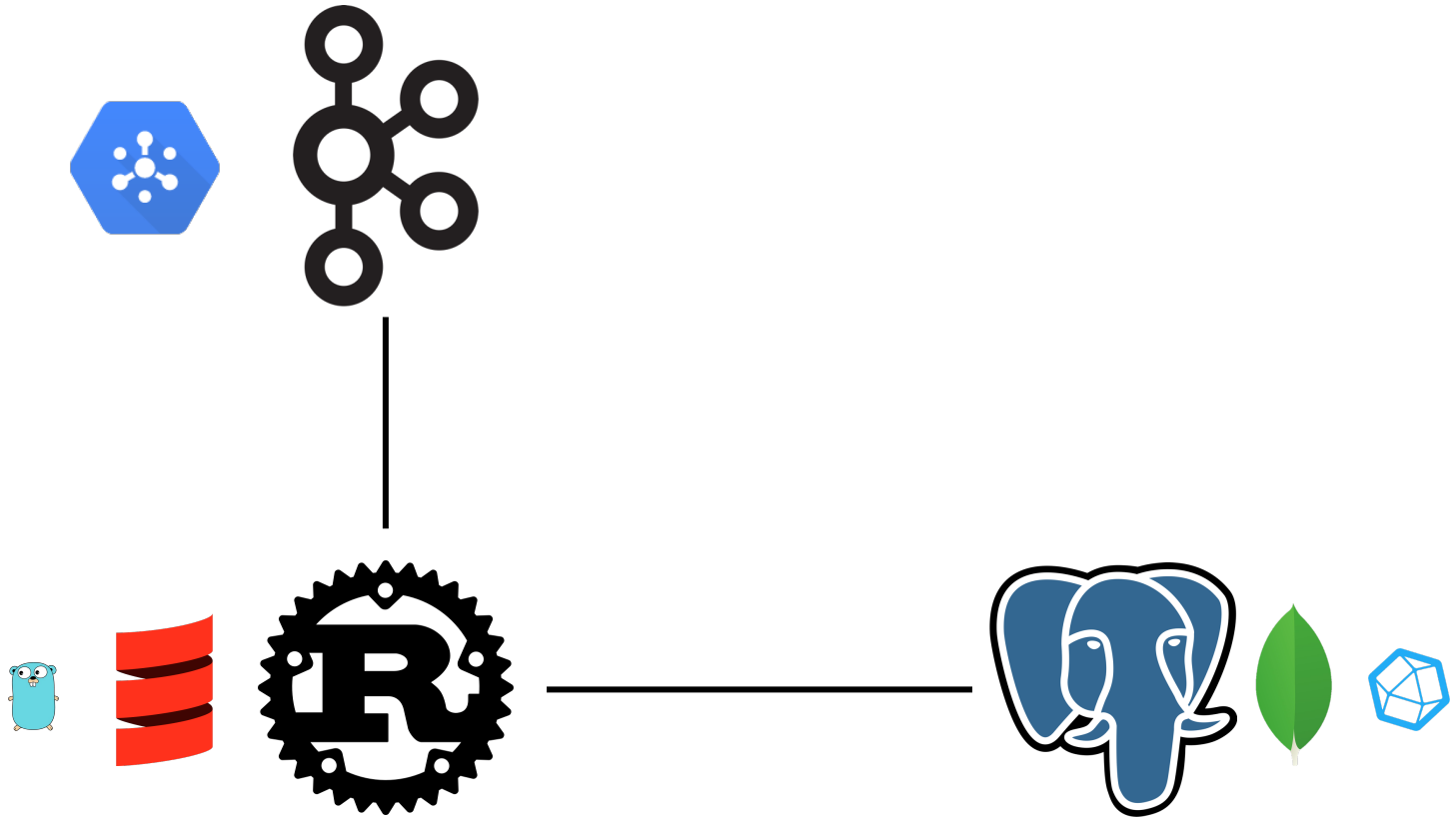


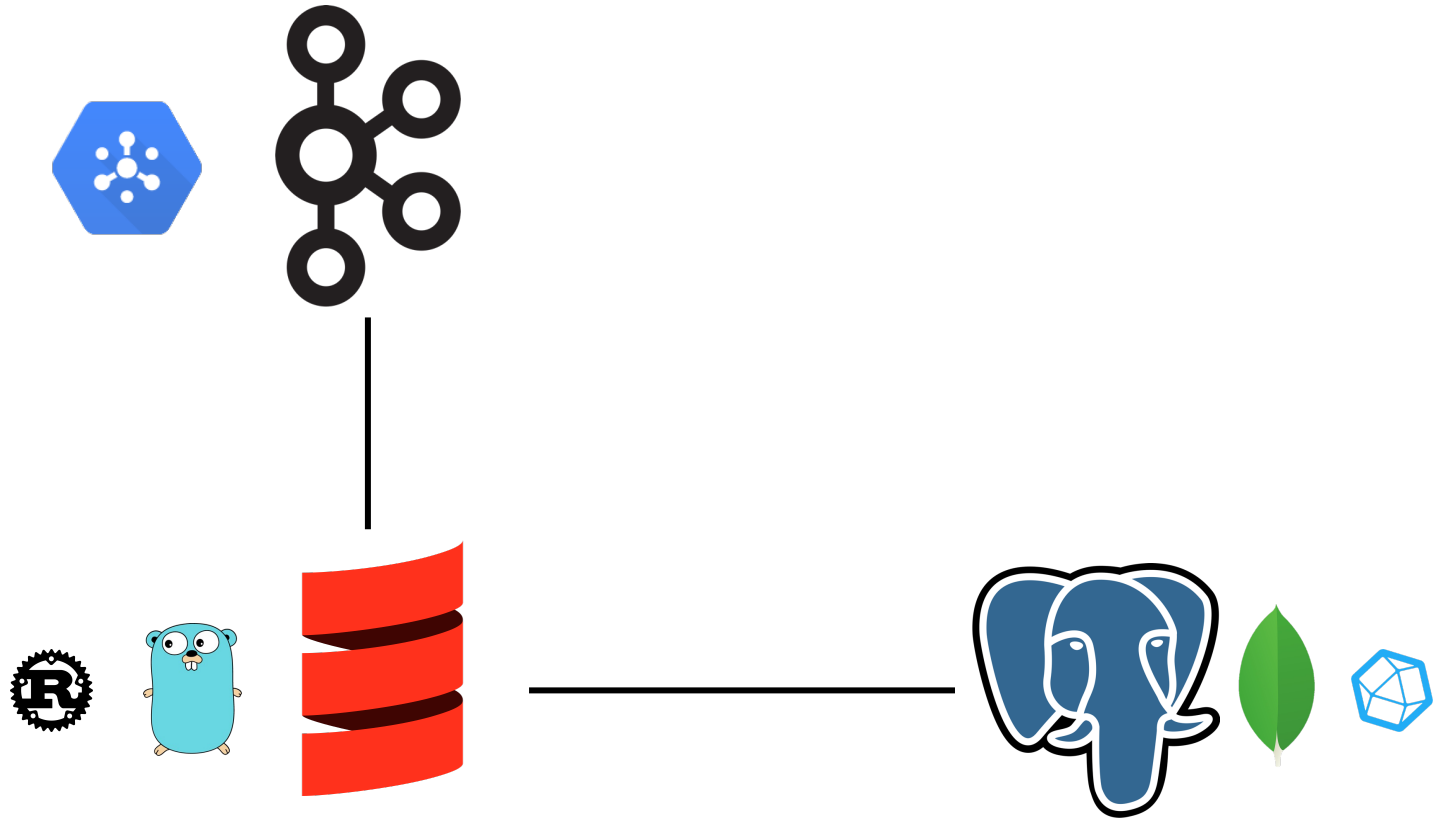


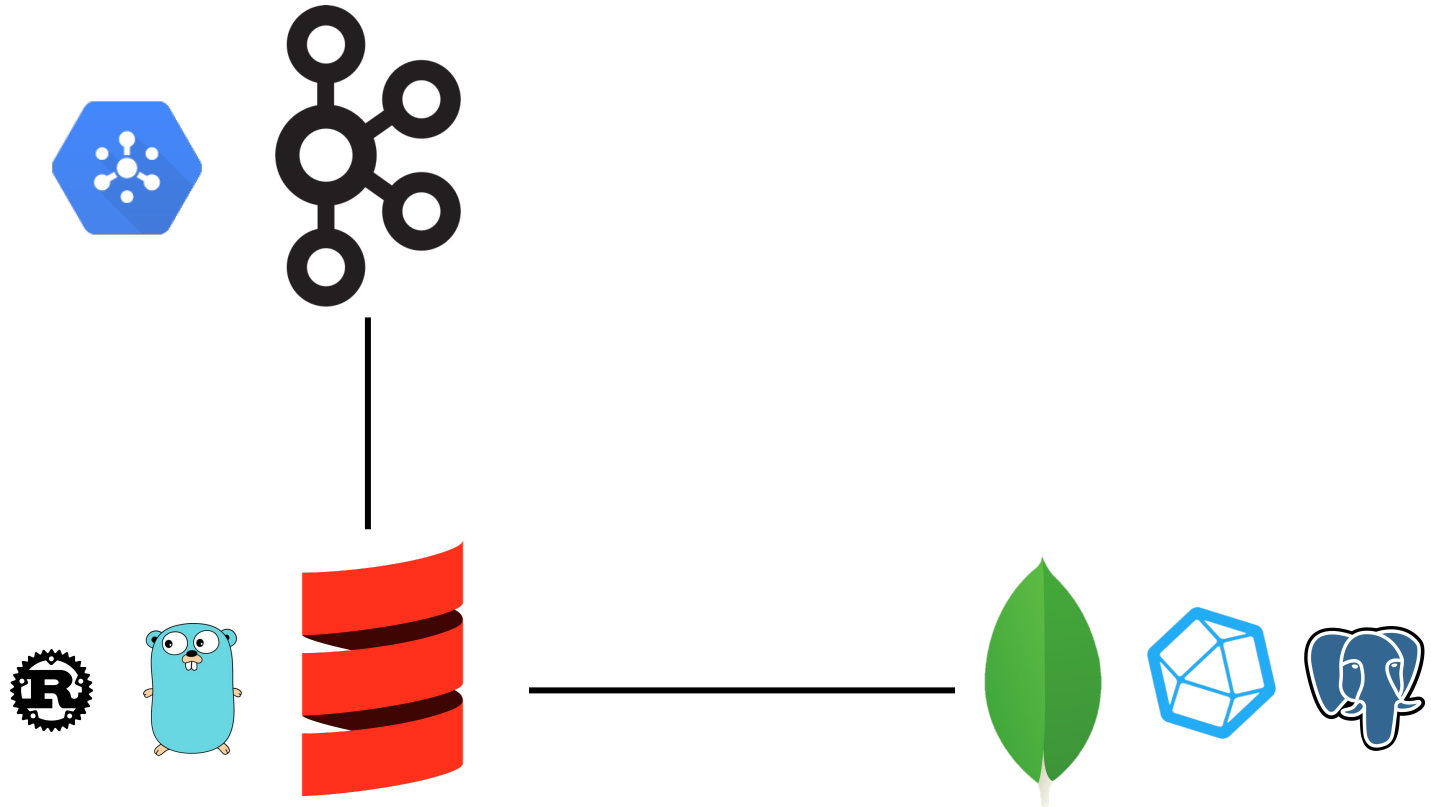


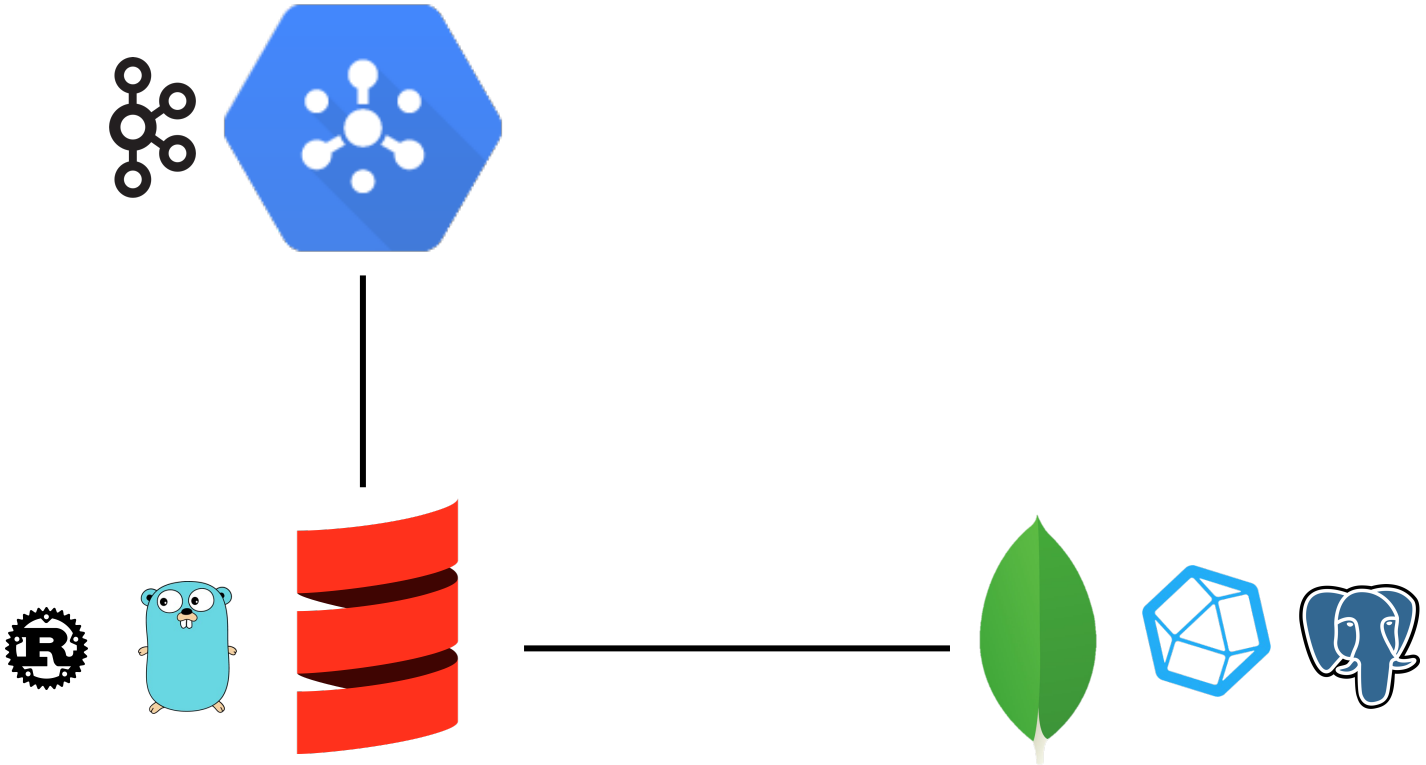


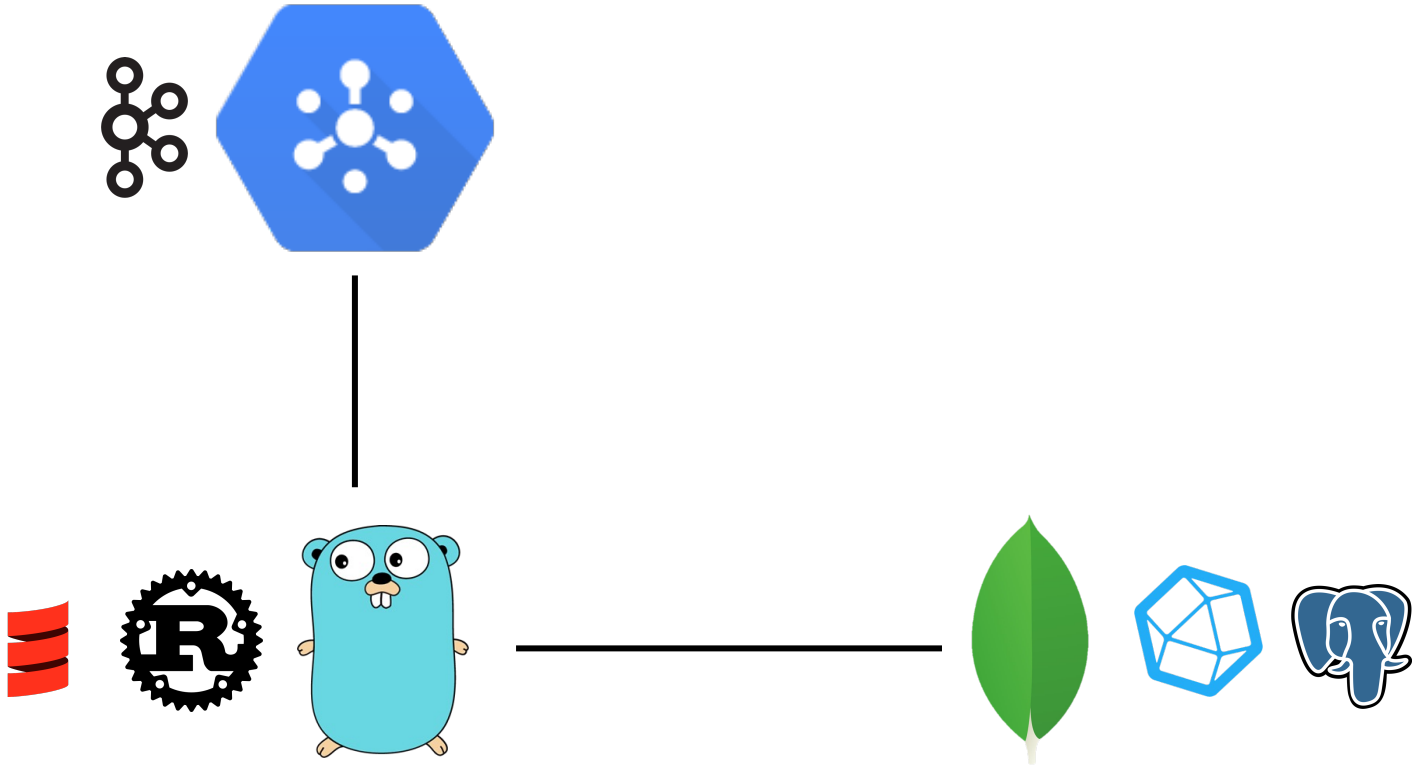


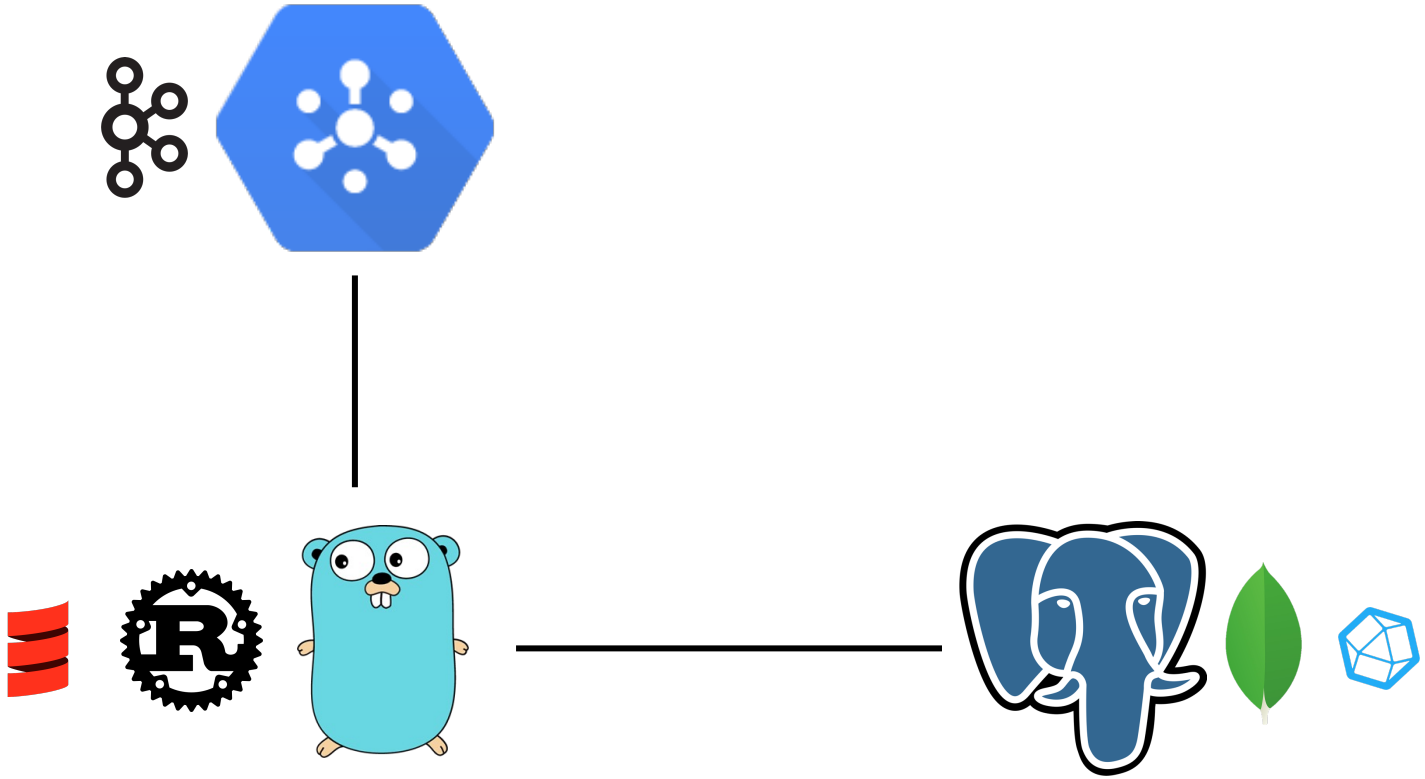


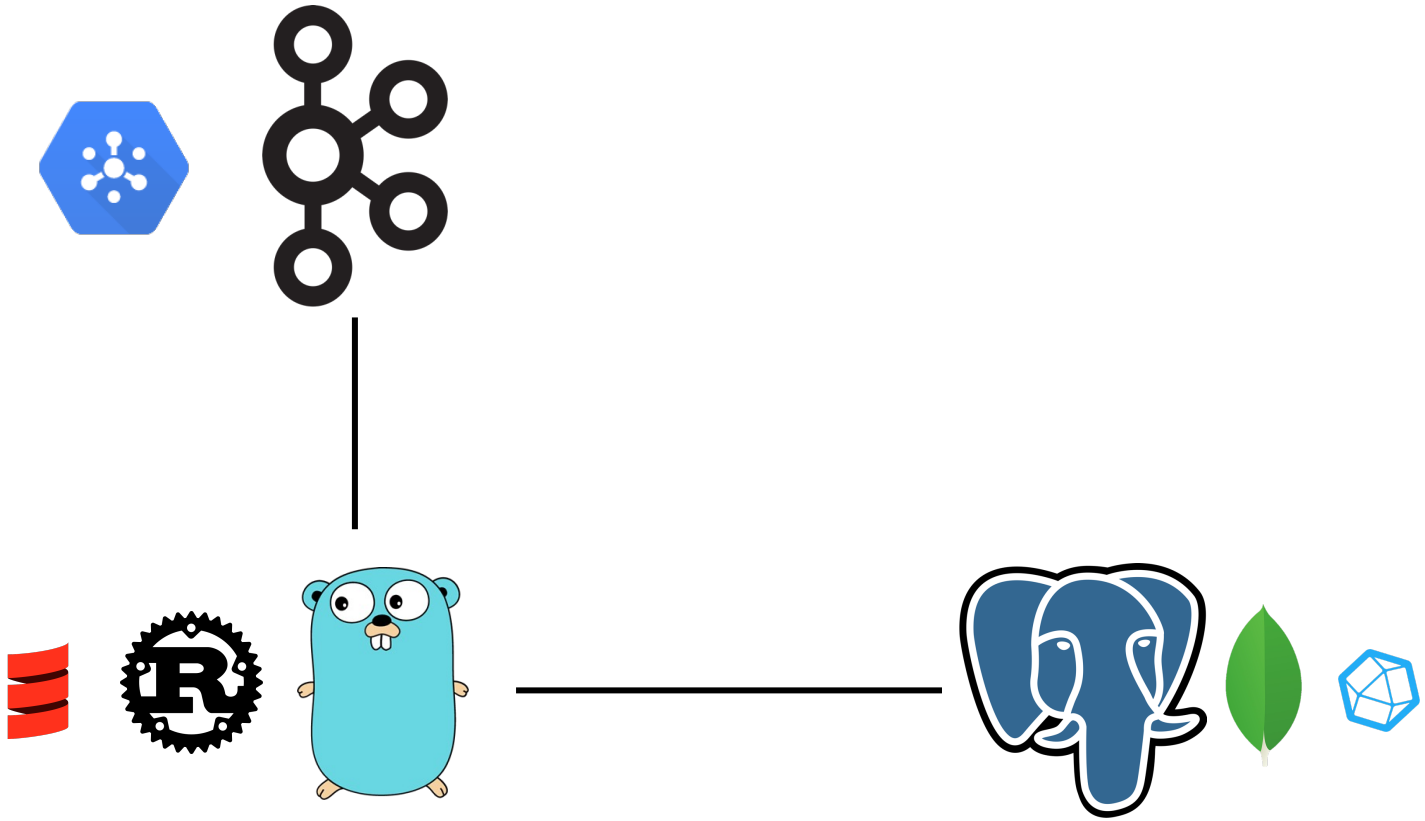




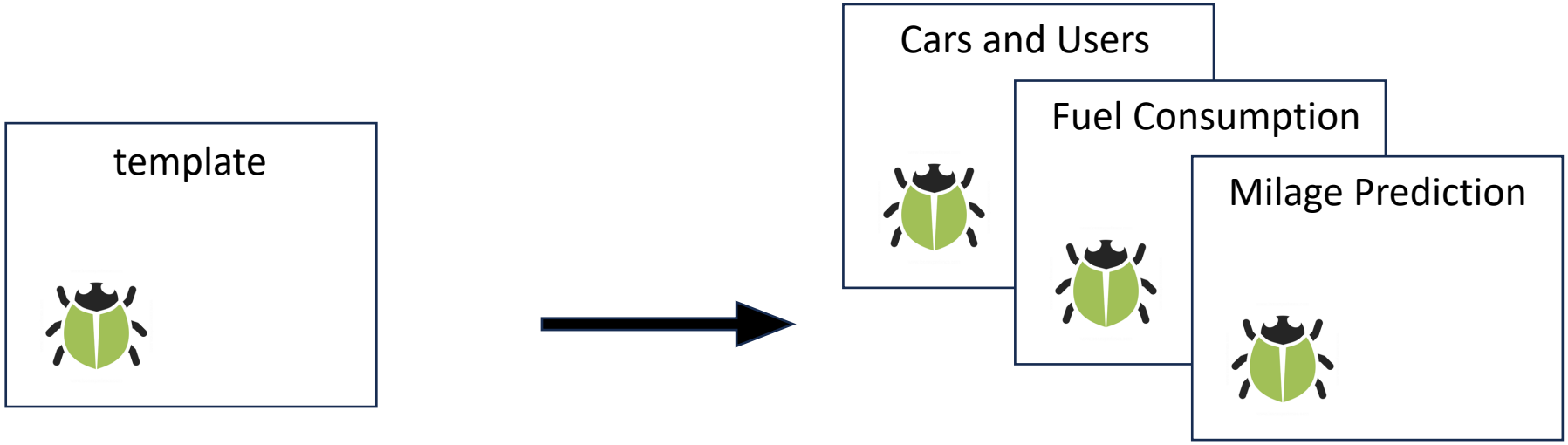








Another day of writing templates ☹



```
pub async fn create_car
```

```
pub async fn manufacture_car
```

Still a proof of concept

All the code used in this presentation is generated by Lightspeed