

# Multitenancy with FastAPI, SQLAlchemy and PostgreSQL

*Sysmagine*

30–39 minutes

---

When building a web application where customers (your tenants) are mostly working with their own data, it is advisable to have a strong separation in how their data is stored and handled. One solution would be to spin up an instance of your software in a VM or docker container for each of your tenants. While this would give you a very strong separation, the solution has a few drawbacks. It gets complicated if you actually want to share some data across tenants, you allocate resources for tenants even if they don't use your app and managing hundreds of VMs or containers is not trivial. In this blog post I want to show you an alternative solution, how you can implement the data separation by using only one FastAPI backend without making your code complicated.

Before we dive into the tutorial, let me give you an overview of the goals we try to achieve:

- Each tenant should get its own set of database tables, with the possibility of shared tables
- Tenant handling should be done implicitly/automatically
- The architecture should make it hard to accidentally access the wrong data
- Support for alembic migrations across all tenants in a single transaction
- Easy way to upgrade an existing code base

## What is multitenancy?

We have already covered some aspects of multitenancy in the introduction, but let me give you some examples of what this looks like in practice. The idea of multitenancy is to give each customer (tenant) the illusion of their own separate workspace, while using the same backend and resources. Real world examples for this are Slack and Notion. You get your own set of users, settings and channels/documents and there is no sign of other tenants in the system - except that you can explicitly share some channels or documents. A counterexample would be GitHub. While you can have your own private repositories, everything is shared by default. There is, for example, only one global list of users and you can invite each user to any project.

The advantage of a multitenancy system, in contrast to a system like GitHub, is that mistakes often have fewer consequences. Sharing a document with the wrong employee or colleague is not as bad as sharing it with a complete stranger. Therefore, a strong separation between your customers can make them feel more safe and encouraged to try things out.

## How to implement multitenancy support

In the following steps I will show you how to make an existing FastAPI app multitenancy ready. Since the changes are not exactly trivial, we won't start from scratch. I assume that your code base looks similar to the one described in the [SQL \(Relational\) Databases](#) FastAPI tutorial and that you have some basic knowledge about FastAPI, SQLAlchemy and Alembic. The database used in the following steps is PostgreSQL.

The same concept should also be compatible with other databases supporting schemas, but I didn't verify this. If you are using MySQL, you are out of luck though. You can still implement multitenancy support, but this would require significant changes to the concept described here.

### Step 1: How to distinguish tenants

The first step in implementing a multitenancy system is to somehow distinguish the tenants in our API endpoints. The most common and simple way is to assign each tenant a different subdomain, like `customer1.myapp.com` and `customer2.myapp.com`. The browser inserts the domain it

wants to access into the `HOST` header field of the request. This was implemented in HTTP/1.1 to support name based virtual hosting, i.e. serving multiple websites using the same IP and port. We can access this value by using the `Request` object provided by FastAPI:

```
from fastapi import FastAPI, Request

app = FastAPI()

@app.get("/")
def root(request: Request):
    return {"Host": request.headers["host"]}
```

We can now test what happens if we try to access the server using different host names (127.0.0.1, localhost) or by overriding the `HOST` value altogether:

```
$ curl http://127.0.0.1:8000
{"Host": "127.0.0.1:8000"}

$ curl http://localhost:8000
{"Host": "localhost:8000"}

curl -H "Host: customer1.myapp.com"
http://127.0.0.1:8000
{"Host": "customer1.myapp.com"}
```

Everything seems to work as expected, we can differentiate tenants in our example code based on the used (sub)domain or IP. This is only for demonstration purposes though, the final solution will be much more elegant than adding code to each API endpoint.

When deploying such an app, making sure that each subdomain is handled by your backend is only part of the solution. You also need a DNS record for that subdomain pointing to the IP of your server. You have basically two options here. You can add an explicit DNS record for each tenant, which can introduce delays and you might need to fiddle around the API of your DNS provider. Or you can instead add a wildcard record `*.myapp.com` that resolves all subdomains to the same IP without further configuration. You can still override the wildcard with a specific record, so you don't lose any flexibility.

The same applies to SSL/TLS certificates. If you need to request a new certificate every time a customer starts a trial, you introduce some delays. It might therefore make sense to use a wildcard certificate here as well. Thanks to Let's Encrypt this is neither complicated nor expensive any more.

## Step 2: Choosing a separation strategy for the data

The real challenge in developing a multitenant application is separating the data in the database and supporting that in your code. There are several ways to achieve this, all with their own pros and cons. Here are the approaches we have analyzed to give you an overview:

### A separate database per tenant

| Pros                                                                                                                                      | Cons                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Easy to delete tenants</li><li>• Strong separation</li><li>• Supported by all databases</li></ul> | <ul style="list-style-type: none"><li>• No support for foreign keys into shared data</li><li>• You cannot use transactions across tenants</li><li>• Replication setups and backups require more work</li></ul> |

### Using a prefix/postfix in table names for tenants

| Pros                                                                                                                                                                                  | Cons                                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Foreign keys into shared data are possible</li><li>• Transactions across tenants are supported</li><li>• Supported by all databases</li></ul> | <ul style="list-style-type: none"><li>• Removing a tenant is difficult, each table needs to be removed</li><li>• No hierarchy, number of tables can grow large</li></ul> |

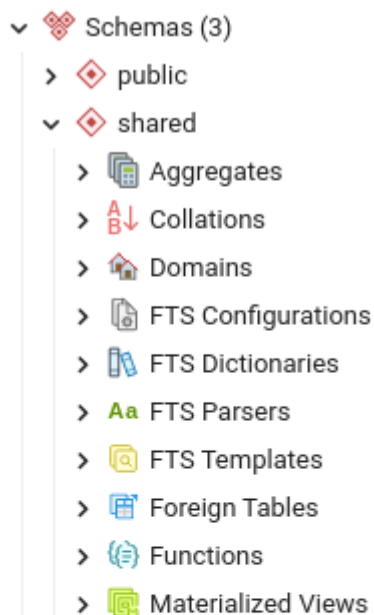
### Adding a tenant column to each table

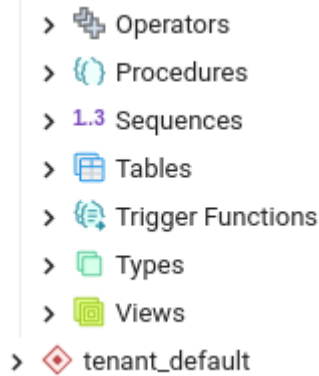
| Pros                                                                                                                                                                     | Cons                                                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Data can easily be shared</li> <li>• Transactions across tenants are supported</li> <li>• Supported by all databases</li> </ul> | <ul style="list-style-type: none"> <li>• <b>Very error prone</b></li> <li>• Removing a tenant is difficult, all affected rows need to be removed</li> <li>• One large index instead of multiple smaller indices</li> </ul> |

## Using a schema per tenant

| Pros                                                                                                                                                                             | Cons                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Foreign keys into shared data are possible</li> <li>• Transactions across tenants are supported</li> <li>• Strong separation</li> </ul> | <ul style="list-style-type: none"> <li>• Not supported by MySQL/MariaDB</li> </ul> |

After comparing all the different solutions, we concluded that schemas are the best option for us. Schemas are an extra hierarchy layer inside databases that can be used to group objects like tables, types or functions together. The only disadvantage is that some databases like MySQL/MariaDB do not have proper support for schemas - which was not an issue for us as we are using PostgreSQL.





If you have used PostgreSQL in the past and opened your database in pgAdmin, you have most likely come across the Schemas list. Usually you have only one entry there: public. All tables you create and queries you run affect the public schema by default. How do you work with other schemas then? Well, the answer is simple. If you run a query like:

Postgres by default translates this to:

```
SELECT * FROM public.mytable;
```

To work with other schemas, you can simply prepend the schema name to the object you want to access or create.

You can also change the default schema or instruct Postgres to search for objects across multiple schemas. I personally avoid using this approach though, because it is easy to forget that you have altered the search path and suddenly work with the wrong data. In this tutorial we want try to achieve the opposite and make it extra hard to access the wrong data using SQLAlchemy. If you want to know more about these commands anyways, take a look [here](#).

For the rest of the tutorial we will use two schemas, `shared` for all the shared data and `tenant_[NAME]` for a tenant with the name `[NAME]`. We are not going to use the `public` schema for one simple reason: This way, if we discover that any of our tables or types end up in the default schema, we have an indicator that we have messed something up :-). The `public` schema should only contain the `alembic_version` table which is used by Alembic to identify the currently deployed version.

### Step 3: Using separate schemas

The next step is to decide for each table whether it should be

tenant specific or shared across tenants. The chances are high that you want to make all or almost all tables tenant specific, but there is at least one table we need to share: The list of tenants.

A very simple tenant database model could look like this:

```
class Tenant(Base):
    __tablename__ = "tenants"

    id = sa.Column("id", sa.Integer,
primary_key=True, nullable=False)
    name = sa.Column("name", sa.String(256),
nullable=False, index=True, unique=True)
    schema = sa.Column("schema",
sa.String(256), nullable=False, unique=True)
    host = sa.Column("host", sa.String(256),
nullable=False, unique=True)

    __table_args__ = ({ "schema": "shared"},)
```

The columns of the tenant table are not very exciting. For each tenant we use an `id` as primary key that we can use as foreign reference in other tables. We store a `name` that will be displayed to the user as well as the name of the `schema` used to store the tenant specific tables. Finally, we need to remember the `host` name under which the tenant is reachable. The most interesting line follows after the column definitions. We use `__table_args__` to set the schema for the table to `shared`. Copy this line to all tables you want to share.

For all remaining tables we can not really specify a fixed schema name as it depends on the tenant making the request. We will therefore use a trick: We use `tenant` as schema name, but instruct SQLAlchemy to dynamically remap it to schema of the current tenant. In analogy to the shared tables we could now add `__table_args__ = ({ "schema": "tenant"},)` to all the affected tables, but there is an easier way. Since the majority of your tables are most likely going to be tenant specific, you can just modify the declarative base and set `tenant` as default schema:

```
metadata = sa.MetaData(schema="tenant")
Base = declarative_base(metadata=metadata)
```

That's it - we have split our tables into tenant specific data and

shared data. Well almost, there is one special case left: Types. If you are using SQLAlchemy Enums in PostgreSQL, we have to decide where to store them. By default they would still end up in the `public` schema, which we try to avoid. We have now two options what to do instead:

- Put all types in the `shared` schema, even if they are not used by the shared tables:

```
status = sa.Column("status", sa.Enum(Status,  
schema="shared"), nullable=False)
```

- Duplicate all tenant specific types and use `shared` only for types required by shared tables:

```
status = sa.Column("status", sa.Enum(Status,  
inherit_schema=True), nullable=False)
```

There is no right or wrong here. Both options have minor pros and cons, but they are mostly a matter of taste. If you want to follow this tutorial as closely as possible, I would recommend using the second option though.

After fixing the enums, we are done modifying our database models. The next steps are to implement the actual tenant switching. Oh, and fix everything we broke by using different schemas ;-).

## Step 4: Initializing a new database

When starting your backend for the first time, you need to initialize the database by creating all necessary tables, types and so on. SQLAlchemy makes this very simple. Just call `create_all` on the metadata of your database models:

```
Base.metadata.create_all(bind=engine)
```

If you tried to execute this command after changing the schemas, you would run into an issue: It would complain that the `shared` and `tenant` schemas do not exist yet. The part with the `shared` schema is easy to fix, but the `tenant` part is more complicated. When we create the database, there is no tenant yet. It therefore doesn't make any sense to create a schema or tables for a tenant. We instead need a way to create only the shared part of our database.

The trick is to write a small function that creates a new metadata

object by filtering the original one and keeping only the objects that are shared:

```
def get_shared_metadata():
    meta = MetaData()
    for table in Base.metadata.tables.values():
        if table.schema != "tenant":
            table.to_metadata(meta)
    return meta
```

Even though we iterate only over the tables, the created metadata will also include all types referenced by those tables. Please note that this implementation will skip any `shared` types that are only referenced by `tenant` tables (important if you decided to put all enums into `shared`). If you depend on this case, you need to extend this function.

With this function in place the database creation would look like this:

```
db.execute(sa.schema.CreateSchema("shared"))
get_shared_metadata().create_all(bind=db)
```

Creating the initial database is an important step, but at some point we also need to migrate our data to a newer version of our database model. For this we will use Alembic. A pre-requirement for Alembic to work is that it initializes its own data when creating the database. This is simple, the only tricky part is to execute everything in a single transaction so that we don't end up with a half initialized database in case of an error. Here is the full code we use to initialize the database, including a check whether it already exists:

```
with engine.begin() as db:
    context = MigrationContext.configure(db)
    if context.get_current_revision() is not None:
        print("Database already exists.")
        return

db.execute(sa.schema.CreateSchema("shared"))
get_shared_metadata().create_all(bind=db)

alembic_config.attributes["connection"] = db
```

```
command.stamp(alembic_config, "head",  
purge=True)
```

After initializing our database, it is time to create our first tenant.

## Step 5: Adding a tenant

So far our database only contains the shared data. The next step is to create the tenant specific tables each time a new tenant is added to our system. For this to work, we need a possibility to remap the hard coded `tenant` schema to the actual schema name of the tenant we want to add. This can be achieved by passing a [schema\\_translate\\_map](#) to the `execution_options` of an SQLAlchemy Connection.

By using this feature of SQLAlchemy we can build a context manager function that returns a database session with any tenant mapping we want:

```
@contextmanager  
def with_db(tenant_schema: Optional[str]):  
    if tenant_schema:  
        schema_translate_map =  
dict(tenant=tenant_schema)  
    else:  
        schema_translate_map = None  
  
    connectable =  
engine.execution_options(schema_translate_map=schema_translate_map)  
  
    try:  
        db = Session(autocommit=False,  
autoflush=False, bind=connectable)  
        yield db  
    finally:  
        db.close()
```

The highlighted lines show how we remap `tenant` to the schema passed via the `tenant_schema` parameter. That is all the “magic” behind our multitenancy system. We simply exchange the mapping based on the tenant accessing our API and we are done. 95% of your existing code will simply work without changes. But more on this later. We first need another utility function to identify all tables we need to create now:

```
def get_tenant_specific_metadata():
    meta = MetaData(schema="tenant")
    for table in Base.metadata.tables.values():
        if table.schema == "tenant":
            table.to_metadata(meta)
    return meta
```

Not surprisingly, this is the exact opposite of the `get_shared_metadata` function from step 4. We need to initialize all tables after all ;-).

Now we have all the dependencies we need to write our `tenant_create` function:

```
def tenant_create(name: str, schema: str, host: str) -> None:
    with with_db(schema) as db:
        tenant = Tenant(
            name=name,
            host=host,
            schema=schema,
        )
        db.add(tenant)

    db.execute(sa.schema.CreateSchema(schema))

    get_tenant_specific_metadata().create_all(bind=db)

    db.commit()
```

The function inserts an entry into our tenant table, creates the schema as well as all tenant specific tables. Everything is done in a single transaction, so that no inconsistent state can be created. There is still some way how you can really mess up your database: If you add a new tenant without applying all database migrations first. Your tenants would have inconsistent database models and you can get all kinds of bugs. To prevent this, we drop in a check to verify that all migrations were applied:

```
def tenant_create(name: str, schema: str, host: str) -> None:
    with with_db(schema) as db:
        context =
MigrationContext.configure(db.connection())
```

```

        script =
alembic.script.ScriptDirectory.from_config(alembic.config.Config('alembic.ini'))
        if context.get_current_revision() != script.get_current_head():
            raise RuntimeError(
                "Database is not up-to-date.
Execute migrations before adding new tenants."
            )

        tenant = Tenant(
            name=name,
            host=host,
            schema=schema,
        )
        db.add(tenant)

db.execute(sa.schema.CreateSchema(schema))

get_tenant_specific_metadata().create_all(bind=db)

db.commit()

```

That is all we need to do to add a new tenant. You might want to make this function accessible via a CLI script so that you can easily add tenants. Removing a tenant is even simpler and can be done in 2 lines: Drop the schema and remove the entry from the tenant tables.

## Step 6: Implementing multitenancy in API endpoints

Let's assume your API function looks like the following code from the FastAPI tutorial. Each tenant has its own user list and we want to return different results depending on the tenant requesting `/users/{user_id}`. What do we have to change?

```

@app.get("/users/{user_id}",
response_model=schemas.User)
def read_user(user_id: int, db: Session =
Depends(get_db)):
    db_user = crud.get_user(db,
user_id=user_id)
    if db_user is None:

```

```
        raise HTTPException(status_code=404,
detail="User not found")
    return db_user
```

The good news is: nothing! We only need to change one of its dependencies - `get_db`. The idea is that `get_db` automatically uses the correct schema remapping based on the tenant making the request. All your queries are then automatically run against the correct tenant. Since you only have to do this once, porting your API endpoints is ridiculously easy.

All we have to do, is to extract the host from the request object, lookup the corresponding tenant schema from the database and call `with_db` with the correct schema:

```
def get_tenant(req: Request) -> Tenant:
    host_without_port =
req.headers["host"].split(":", 1)[0]

    with with_db(None) as db:
        tenant =
db.query(Tenant).filter(Tenant.host==host_without

    if tenant is None:
        raise TenantNotFoundError()

    return tenant

def get_db(tenant: Tenant =
Depends(get_tenant)):
    with with_db(tenant.schema) as db:
        yield db
```

Before you get too excited though, the shown solution has one drawback. We need to use two database connections since SQLAlchemy does not allow us to change the mapping after we established the connection. Querying the tenant from the database for each incoming request is also not super efficient. We can easily fix this by implementing a simple in-memory cache for `get_tenant`. Since a tenant's values should rarely change, you don't need anything fancy. The task is therefore left to my motivated readers :-). Just be aware that the function needs to be multithreading safe.

The code will also not run without modifications since it uses a

self-defined `TenantNotFoundError` Exception. This exception is thrown if the tenant does not exist and then caught again using a FastAPI [exception\\_handler](#) to turn it into a 403 Forbidden HTTP error. The HTTP error in combination with a specific error type is then detected by the frontend to show an appropriate error page.

In some cases the database mapping itself is not sufficient. You may want to know more information about the tenant itself, e.g. to display their name in the frontend. That is easy to achieve, you can directly depend on `get_tenant` as well. FastAPI makes sure that `get_tenant` is only executed once, even if several functions depend on it.

```
@app.get("/tenant",
response_model=schemas.TenantInfo)
def get_tenant(db: Session = Depends(get_db),
tenant: Tenant = Depends(get_tenant)):
    # Do something with db
    return tenant
```

If you make your tenant model complex, for example by adding relationships, or if you want to modify the tenant values, you might run into an issue. The database connection is closed after querying the tenant object, which expunges it from the database session. This means you can no longer resolve any relationships (the database connection is gone) and changes to the tenant object are no longer tracked. To work around this, make sure that `get_tenant` preloads all values that you need. If you want to change the tenant values, query the object again and modify it while keeping the database connection open.

## Step 7: Migrations

So far our journey has been quite smooth and we haven't encountered any major obstacles. Now you better buckle up as we need to implement database migrations ;-).

The go-to solution for SQLAlchemy database migrations is [Alembic](#). It compares your SQLAlchemy models against your database definitions and then generates a migration script to upgrade your tables. Even if you had a working alembic configuration so far, it won't work any more. Your SQLAlchemy definition states that all tenant specific tables to reside under the

tenant schema, which is just a placeholder for a real tenant. How do we fix this?

For Alembic to work, it needs to be able to compare the database models against some tables. We therefore need at least one tenant in our backend to compare against. You could now make your life easy and assign that tenant the schema `tenant` to match the SQLAlchemy definitions. We decided against this solution because having a tenant with the schema `tenant` in our development databases could hide potential bugs where our app fails to properly remap the schema name. We instead decided to use a special tenant on our development systems with the schema `tenant_default` and inform alembic about this mapping.

To implement the schema remapping from `tenant` to `tenant_default` in alembic, open `alembic/env.py` in your project and add the highlighted lines:

```
def run_migrations_online():
    """Run migrations in 'online' mode.

    In this scenario we need to create an
    Engine
    and associate a connection with the
    context.
    """

    translated =
    MetaData(naming_convention=naming_convention)

    def translate_schema(table, to_schema,
constraint, referred_schema):
        # pylint: disable=unused-argument
        return to_schema

    for table in Base.metadata.tables.values():
        table.tometadata(
            translated,
            schema="tenant_default" if
table.schema == "tenant" else table.schema,
referred_schema_fn=translate_schema,
        )
```

```

with engine.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=translated,
        compare_type=True,
        transaction_per_migration=True,
        include_schemas=True,
    )

    with context.begin_transaction():
        context.run_migrations()

```

After adding the translation logic, you should be able to generate new migration scripts again. Let us take look at how such a generated script would look like:

```

def upgrade():
    op.add_column(
        "users",
        sa.Column("time_expires",
sa.DateTime(timezone=True), nullable=True),
        schema="tenant_default",
    )

def downgrade():
    op.drop_column("users", "time_expires",
schema="tenant_default")

```

On first sight the script looks great, but there is still one big issue left. It tries to apply the changes to the potentially non-existing schema `tenant_default`. What we actually want to do is apply this change to **every** tenant in our database. How can we achieve this?

A somewhat simple solution we came up with was to create a function decorator that calls a function for every tenant in our database. For this we created the script `alembic/tenant.py`:

```

import functools
from typing import Callable

from typeguard import typechecked
from alembic import op

```

```

@typechecked
def for_each_tenant_schema(func: Callable) ->
Callable:
    @functools.wraps(func)
    def wrapped():
        schemas = op.get_bind().execute("SELECT
schema FROM shared.tenants").fetchall()
        for (schema,) in schemas:
            func(schema)

    return wrapped

```

We can now import `for_each_tenant_schema` from a migration script and put all tenant specific upgrades into a function using this decorator. Let's apply this solution to our example migration:

```

# More code ...

from .tenant import for_each_tenant_schema

# More code ...

@for_each_tenant_schema
def upgrade(schema: str):
    op.add_column(
        "users",
        sa.Column("time_expires",
sa.DateTime(timezone=True), nullable=True),
        schema=schema,
    )

@for_each_tenant_schema
def downgrade(schema: str):
    op.drop_column("users", "time_expires",
schema=schema)

```

Since all our modifications are tenant specific, we can add the decorator simply to the `upgrade` and `downgrade` functions. Sadly this change alone is not sufficient, we still have to manually replace every occurrence of `tenant_default` with

our function parameter `schema`.

Manually adding the decorator every time we need it is a bit of a chore. Luckily, there is a way to make this the default logic of the generated script. Just open `alembic/script.py.mako` and add the highlighted lines:

```
"""${message}

Revision ID: ${up_revision}
Revises: ${down_revision | comma,n}
Create Date: ${create_date}

"""
from alembic import op
import sqlalchemy as sa
${imports if imports else ""}

from .tenant import for_each_tenant_schema

# revision identifiers, used by Alembic.
revision = ${repr(up_revision)}
down_revision = ${repr(down_revision)}
branch_labels = ${repr(branch_labels)}
depends_on = ${repr(depends_on)}

@for_each_tenant_schema
def upgrade(schema: str):
    preparer =
sa.sql.compiler.IdentifierPreparer(op.get_bind()
    schema_quoted =
preparer.format_schema(schema)

    ${upgrades if upgrades else "pass"}

@for_each_tenant_schema
def downgrade(schema: str):
    preparer =
sa.sql.compiler.IdentifierPreparer(op.get_bind()
    schema_quoted =
```

```
preparer.format_schema(schema)
```

```
    ${downgrades if downgrades else "pass"}
```

If you look closely I didn't only add the decorator but also code to compute the variable `schema_quoted`. This variable contains the escaped schema name that can be used in queries like `op.execute(f"DROP TYPE {schema_quoted}.myenum")`. If your migration doesn't need any text SQL statements, you can just remove it again from the generated script.

With these changes in place, you should be able to generate working migrations again. The process is a bit annoying as you have to fix up every generated migration script, but it is easily done using search & replace. Many auto-generated scripts needs some tweaking anyways, since Alembic can not yet handle all types of changes yet, like adding a value to an existing PostgreSQL enum. Nevertheless, this is the most error prone step of adding multitenancy support. Improving this situation would most probably require patching Alembic.

One last tip: To prevent introducing any bugs when manually changing the migration scripts, we added a step to our CI which upgrades a database from an old version and then compares the schema<sup>[1]</sup> against a freshly initialized database. To be extra safe, both databases contain two tenants. If the schema<sup>[1]</sup> differs, the test fails. This doesn't guarantee that we didn't mess up the data during an upgrade but it at least ensures that the table definitions, types, ... are the same. This approach can also help to detect cases where Alembic couldn't handle some changes.

[1]: The term schema is ambiguous in the context of databases. In this case we refer to the SQL definitions that make up tables, types and so on.

## Step 8: Upgrading an existing database

If you don't have a production database yet, you can most likely skip this step. Migrating all tables and types in your database is a bit tedious and most probably not justified if you can just recreate your database. In order to keep your data, follow these steps:

1. Create the schema `tenant_default` for your “existing” tenant
2. Move all tenant specific tables and types into this new schema
3. Create the `shared` schema
4. Move the remaining tables and types to `shared`
5. Create the `tenant` table in `shared`
6. Insert the new tenant into your `tenant` table

The slightly bad news is that there is no good way to auto-generate the first steps. You can make your life a bit easier though by creating two migrations. First, a semi-manually written migration to move everything into the correct schemas (step 1-4) and then let alembic generate the code for step 5. By using this approach you will also notice if you missed moving a table or type. These objects would show up as addition in the auto-generated script, which should only contain changes related to the tenant table.

To write the migration for the steps 1-4, you first need a list of all your existing tables

```
SELECT
    "table_name"
FROM
    information_schema.tables
WHERE
    table_schema = 'public'
```

and enums:

```
SELECT
    t.typname as enum_name,
    e.enumlabel as enum_value
FROM
    pg_type t
    JOIN pg_enum e on t.oid = e.enumtypid
    JOIN pg_catalog.pg_namespace n ON n.oid =
t.typnamespace
WHERE
    n.nspname = 'public'
```

If you have any further database objects, gather them as well. Now create a migration to move them all to their new schemas:

```
def upgrade():
```

```
op.execute("CREATE SCHEMA shared")

op.execute("ALTER TABLE public.table1 SET
SCHEMA shared")
op.execute("ALTER TABLE public.table2 SET
SCHEMA shared")

op.execute("CREATE SCHEMA tenant_default")

op.execute("ALTER TABLE public.table3 SET
SCHEMA tenant_default")
op.execute("ALTER TABLE public.table4 SET
SCHEMA tenant_default")

op.execute("ALTER TYPE public.type1 SET
SCHEMA tenant_default")
```

The chances are high that you don't have any shared tables yet. In this case you can just use search & replace to write the migration for you based on the list of database objects.

After applying the migration, let Alembic generate the required upgrade statements for the new `tenant` table. To finish the transition to our multitenant system, we also need to add a tenant record to our newly created tables. You could solved this by inserting the corresponding row directly as part of the migration script. The only issue with this solutions is that you most probably can not get the `host` value to match your development systems as well as your production system. It is therefore up to you whether you want include something like the following snippet or run the SQL query manually on all affected systems.

```
query = sa.text(
    "INSERT INTO shared.tenants (name, schema,
host) "
    "VALUES (:name, :schema, :host)"
).bindparams(name="default",
schema="tenant_default", host="127.0.0.1")
op.execute(query)
```

In either case, you are done. You should be able to access your existing tenant and add new tenants.

## Summary

Making an existing FastAPI app multitenancy ready isn't as complicated as you might expect. Based on all the code examples and long explanations, it may look difficult at first glance, but you probably need to change less than 100 lines to make it work. The best aspect of the described solution is that it will just work for most newly written API endpoints - no special care is required. The only tricky part is the Alembic migrations. They require a bit of fiddling to work correctly, but the provided code changes and templates should hopefully reduce the pain quite a bit.

I hope this blog post has been useful to you and that you can save some time when you need to implement multitenancy support. Also, let me know if you found any errors in the instructions or if you know a better way to solve certain aspects. Just send a twitter DM to [@mmueller2012](#). You are welcome to follow me as well if you don't want to miss any new blog posts.