Below is a complete, working MVP codebase you can drop into a new repo. It’s an API‑first service (FastAPI + SQLAlchemy) that:

* Creates and manages business partners
* Issues invoices in UAE‑friendly fields (incl. VAT/TRN), stores them, and tracks lifecycle statuses
* Validates required fields and basic UAE/PEPPOL business rules
* Builds a PEPPOL‑BIS‑style XML payload (simplified) and sends via an **accredited provider API** (adapter pattern)
* Provides an **API‑key** auth mechanism, structured logging, and a clean service layer
* Ships with Dockerfile, tests (pytest), and a sample CI config

# 🗂️ Repository Layout

beam/  
├─ app/  
│ ├─ \_\_init\_\_.py  
│ ├─ main.py  
│ ├─ config.py  
│ ├─ auth.py  
│ ├─ db.py  
│ ├─ models.py  
│ ├─ schemas.py  
│ ├─ enums.py  
│ ├─ services/  
│ │ ├─ \_\_init\_\_.py  
│ │ ├─ partner\_service.py  
│ │ └─ invoice\_service.py  
│ ├─ peppol/  
│ │ ├─ \_\_init\_\_.py  
│ │ ├─ provider\_client.py  
│ │ ├─ xml\_builder.py  
│ │ └─ validators.py  
│ ├─ utils/  
│ │ ├─ \_\_init\_\_.py  
│ │ ├─ crypto.py  
│ │ └─ logging.py  
│ └─ middleware/  
│ ├─ \_\_init\_\_.py  
│ └─ request\_context.py  
├─ tests/  
│ ├─ \_\_init\_\_.py  
│ └─ test\_invoice\_flow.py  
├─ .env.example  
├─ .gitignore  
├─ Dockerfile  
├─ pyproject.toml  
├─ requirements.txt  
└─ README.md

## app/**init**.py

# Empty package marker

## app/enums.py

from enum import Enum  
  
class InvoiceStatus(str, Enum):  
 DRAFT = "DRAFT"  
 VALIDATED = "VALIDATED"  
 SENT = "SENT"  
 DELIVERED = "DELIVERED"  
 REJECTED = "REJECTED"  
  
class Currency(str, Enum):  
 AED = "AED"  
  
class Language(str, Enum):  
 EN = "en"  
 AR = "ar"  
  
class Role(str, Enum):  
 SUPERADMIN = "SUPERADMIN"  
 OWNER = "OWNER"  
 ADMIN = "ADMIN"  
 USER = "USER"  
  
class OrgStatus(str, Enum):  
 PENDING = "PENDING"  
 APPROVED = "APPROVED"  
  
class VATReturnStatus(str, Enum):  
 DRAFT = "DRAFT"  
 FILED = "FILED"

python from enum import Enum

class InvoiceStatus(str, Enum): DRAFT = “DRAFT” VALIDATED = “VALIDATED” SENT = “SENT” DELIVERED = “DELIVERED” REJECTED = “REJECTED”

class Currency(str, Enum): AED = “AED”

class Language(str, Enum): EN = “en” AR = “ar”

class Role(str, Enum): SUPERADMIN = “SUPERADMIN” OWNER = “OWNER” ADMIN = “ADMIN” USER = “USER”

class OrgStatus(str, Enum): PENDING = “PENDING” APPROVED = “APPROVED”

python  
from enum import Enum  
  
class InvoiceStatus(str, Enum):  
 DRAFT = "DRAFT"  
 VALIDATED = "VALIDATED"  
 SENT = "SENT"  
 DELIVERED = "DELIVERED"  
 REJECTED = "REJECTED"  
  
class Currency(str, Enum):  
 AED = "AED"  
  
class Language(str, Enum):  
 EN = "en"  
 AR = "ar"

## app/config.py

from pydantic import BaseSettings, AnyHttpUrl, Field  
from typing import Optional  
  
class Settings(BaseSettings):  
 APP\_NAME: str = "Beam"  
 API\_V1\_PREFIX: str = "/api/v1"  
 API\_KEY: str = Field("dev-key", description="Legacy API key (optional)")  
  
 DB\_URL: str = "sqlite+aiosqlite:///./beam.db"  
  
 # PEPPOL provider settings (adapter will use these)  
 PROVIDER\_BASE\_URL: Optional[AnyHttpUrl] = None  
 PROVIDER\_API\_KEY: Optional[str] = None  
 PROVIDER\_TIMEOUT\_SECONDS: int = 20  
  
 # UBL validation  
 UBL\_XSD\_PATH: Optional[str] = None # e.g., ./xsd/maindoc/UBL-Invoice-2.1.xsd  
  
 # UAE / Compliance  
 UAE\_REQUIRE\_LOCAL\_STORAGE: bool = True  
 DEFAULT\_LANGUAGE: str = "en"  
  
 # Auth (JWT/OAuth2)  
 JWT\_SECRET: str = Field("dev-secret", description="Change in production")  
 JWT\_ALGORITHM: str = "HS256"  
 ACCESS\_TOKEN\_EXPIRE\_MINUTES: int = 60  
  
 # Admin/Dev  
 AUTO\_APPROVE\_SIGNUPS: bool = True # keep tests/dev simple; disable in prod  
  
 # Digital Signing / Hash Chain  
 SIGNING\_PRIVATE\_KEY\_PEM: Optional[str] = None # paste PEM or mount path contents  
 SIGNING\_CERT\_SERIAL: Optional[str] = None # optional certificate serial for audit trail  
 HASH\_ALG: str = "sha256"  
  
 class Config:  
 env\_file = ".env"  
  
settings = Settings()

python from pydantic import BaseSettings, AnyHttpUrl, Field from typing import Optional

class Settings(BaseSettings): APP\_NAME: str = “Beam” API\_V1\_PREFIX: str = “/api/v1” API\_KEY: str = Field(“dev-key”, description=“Simple API key for demo”)

DB\_URL: str = "sqlite+aiosqlite:///./beam.db"  
  
# PEPPOL provider settings (adapter will use these)  
PROVIDER\_BASE\_URL: Optional[AnyHttpUrl] = None  
PROVIDER\_API\_KEY: Optional[str] = None  
PROVIDER\_TIMEOUT\_SECONDS: int = 20  
  
# UAE / Compliance  
UAE\_REQUIRE\_LOCAL\_STORAGE: bool = True  
DEFAULT\_LANGUAGE: str = "en"  
  
class Config:  
 env\_file = ".env"

settings = Settings()

## app/db.py  
```python  
from sqlalchemy.ext.asyncio import create\_async\_engine, async\_sessionmaker, AsyncSession  
from sqlalchemy.orm import declarative\_base  
from .config import settings  
  
engine = create\_async\_engine(settings.DB\_URL, echo=False, future=True)  
AsyncSessionLocal = async\_sessionmaker(engine, expire\_on\_commit=False, class\_=AsyncSession)  
Base = declarative\_base()  
  
async def get\_session() -> AsyncSession:  
 async with AsyncSessionLocal() as session:  
 yield session

## app/models.py

from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from sqlalchemy import String, Integer, ForeignKey, DateTime, Enum as SAEnum, Text, Float, UniqueConstraint  
from datetime import datetime  
from .db import Base  
from .enums import InvoiceStatus, Role, OrgStatus, VATReturnStatus  
  
class Organization(Base):  
 \_\_tablename\_\_ = "organizations"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 name: Mapped[str] = mapped\_column(String(255), unique=True, nullable=False)  
 status: Mapped[OrgStatus] = mapped\_column(SAEnum(OrgStatus), default=OrgStatus.APPROVED)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 users: Mapped[list["Membership"]] = relationship(back\_populates="organization")  
 partners: Mapped[list["Partner"]] = relationship(back\_populates="organization")  
 invoices: Mapped[list["Invoice"]] = relationship(back\_populates="organization")  
  
class User(Base):  
 \_\_tablename\_\_ = "users"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 email: Mapped[str] = mapped\_column(String(255), unique=True, index=True)  
 hashed\_password: Mapped[str] = mapped\_column(String(255))  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 memberships: Mapped[list["Membership"]] = relationship(back\_populates="user")  
  
class Membership(Base):  
 \_\_tablename\_\_ = "memberships"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 user\_id: Mapped[int] = mapped\_column(ForeignKey("users.id"))  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"))  
 role: Mapped[Role] = mapped\_column(SAEnum(Role), default=Role.USER)  
  
 user: Mapped[User] = relationship(back\_populates="memberships")  
 organization: Mapped[Organization] = relationship(back\_populates="users")  
  
 \_\_table\_args\_\_ = (UniqueConstraint("user\_id", "org\_id", name="uq\_user\_org"),)  
  
class Partner(Base):  
 \_\_tablename\_\_ = "partners"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 name: Mapped[str] = mapped\_column(String(255), nullable=False)  
 legal\_name: Mapped[str] = mapped\_column(String(255), nullable=True)  
 trn: Mapped[str] = mapped\_column(String(32), nullable=False, index=True)  
 peppol\_participant\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 country\_code: Mapped[str] = mapped\_column(String(2), default="AE", nullable=False)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 organization: Mapped[Organization] = relationship(back\_populates="partners")  
 outbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="seller", foreign\_keys="Invoice.seller\_id")  
 inbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="buyer", foreign\_keys="Invoice.buyer\_id")  
  
class Invoice(Base):  
 \_\_tablename\_\_ = "invoices"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 number: Mapped[str] = mapped\_column(String(64), nullable=False, unique=True)  
  
 invoice\_type: Mapped[str] = mapped\_column(String(32), default="SALE") # InvoiceType  
  
 seller\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
 buyer\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
  
 issue\_date: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
 due\_days: Mapped[int] = mapped\_column(Integer, default=30)  
  
 currency: Mapped[str] = mapped\_column(String(3), default="AED")  
  
 subtotal: Mapped[float] = mapped\_column(Float, default=0.0)  
 vat\_amount: Mapped[float] = mapped\_column(Float, default=0.0)  
 total: Mapped[float] = mapped\_column(Float, default=0.0)  
  
 status: Mapped[InvoiceStatus] = mapped\_column(SAEnum(InvoiceStatus), default=InvoiceStatus.DRAFT)  
  
 items\_json: Mapped[str] = mapped\_column(Text, default="[]")  
 meta\_json: Mapped[str] = mapped\_column(Text, default="{}")  
  
 provider\_message\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 provider\_status: Mapped[str] = mapped\_column(String(64), nullable=True)  
  
 # --- Signing & Hash Chain ---  
 prev\_hash: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 curr\_hash: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 signature\_b64: Mapped[str] = mapped\_column(Text, nullable=True)  
 signing\_cert\_serial: Mapped[str] = mapped\_column(String(128), nullable=True)  
  
 organization: Mapped[Organization] = relationship(back\_populates="invoices")  
 seller: Mapped[Partner] = relationship(back\_populates="outbound\_invoices", foreign\_keys=[seller\_id])  
 buyer: Mapped[Partner] = relationship(back\_populates="inbound\_invoices", foreign\_keys=[buyer\_id])  
  
# --- Basic Accounting ---  
class AccountType(str):  
 ASSET = "ASSET"  
 LIABILITY = "LIABILITY"  
 EQUITY = "EQUITY"  
 REVENUE = "REVENUE"  
 EXPENSE = "EXPENSE"  
  
class Account(Base):  
 \_\_tablename\_\_ = "accounts"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 code: Mapped[str] = mapped\_column(String(32), index=True)  
 name: Mapped[str] = mapped\_column(String(255))  
 type: Mapped[str] = mapped\_column(String(16))  
  
class JournalEntry(Base):  
 \_\_tablename\_\_ = "journal\_entries"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 ref: Mapped[str] = mapped\_column(String(64))  
 memo: Mapped[str] = mapped\_column(String(255), default="")  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
class JournalLine(Base):  
 \_\_tablename\_\_ = "journal\_lines"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 entry\_id: Mapped[int] = mapped\_column(ForeignKey("journal\_entries.id"), index=True)  
 account\_id: Mapped[int] = mapped\_column(ForeignKey("accounts.id"), index=True)  
 debit: Mapped[float] = mapped\_column(Float, default=0.0)  
 credit: Mapped[float] = mapped\_column(Float, default=0.0)  
  
class Payment(Base):  
 \_\_tablename\_\_ = "payments"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 invoice\_id: Mapped[int] = mapped\_column(ForeignKey("invoices.id"), index=True)  
 amount: Mapped[float] = mapped\_column(Float)  
 method: Mapped[str] = mapped\_column(String(64), default="bank")  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
# --- VAT Returns ---  
class VATReturn(Base):  
 \_\_tablename\_\_ = "vat\_returns"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 period\_start: Mapped[datetime] = mapped\_column(DateTime, index=True)  
 period\_end: Mapped[datetime] = mapped\_column(DateTime, index=True)  
 output\_vat: Mapped[float] = mapped\_column(Float, default=0.0)  
 input\_vat: Mapped[float] = mapped\_column(Float, default=0.0)  
 net\_vat: Mapped[float] = mapped\_column(Float, default=0.0)  
 status: Mapped[VATReturnStatus] = mapped\_column(SAEnum(VATReturnStatus), default=VATReturnStatus.DRAFT)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

# — Basic Accounting —

class AccountType(str): ASSET = “ASSET” LIABILITY = “LIABILITY” EQUITY = “EQUITY” REVENUE = “REVENUE” EXPENSE = “EXPENSE”

class Account(Base): **tablename** = “accounts” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) org\_id: Mapped[int] = mapped\_column(ForeignKey(“organizations.id”), index=True) code: Mapped[str] = mapped\_column(String(32), index=True) name: Mapped[str] = mapped\_column(String(255)) type: Mapped[str] = mapped\_column(String(16))

class JournalEntry(Base): **tablename** = “journal\_entries” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) org\_id: Mapped[int] = mapped\_column(ForeignKey(“organizations.id”), index=True) ref: Mapped[str] = mapped\_column(String(64)) memo: Mapped[str] = mapped\_column(String(255), default=““) created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

class JournalLine(Base): **tablename** = “journal\_lines” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) entry\_id: Mapped[int] = mapped\_column(ForeignKey(“journal\_entries.id”), index=True) account\_id: Mapped[int] = mapped\_column(ForeignKey(“accounts.id”), index=True) debit: Mapped[float] = mapped\_column(Float, default=0.0) credit: Mapped[float] = mapped\_column(Float, default=0.0)

class Payment(Base): **tablename** = “payments” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) org\_id: Mapped[int] = mapped\_column(ForeignKey(“organizations.id”), index=True) invoice\_id: Mapped[int] = mapped\_column(ForeignKey(“invoices.id”), index=True) amount: Mapped[float] = mapped\_column(Float) method: Mapped[str] = mapped\_column(String(64), default=“bank”) created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

# — VAT Returns (MVP) —

class VATReturn(Base): **tablename** = “vat\_returns” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) org\_id: Mapped[int] = mapped\_column(ForeignKey(“organizations.id”), index=True) period\_start: Mapped[datetime] = mapped\_column(DateTime, index=True) period\_end: Mapped[datetime] = mapped\_column(DateTime, index=True) output\_vat: Mapped[float] = mapped\_column(Float, default=0.0) input\_vat: Mapped[float] = mapped\_column(Float, default=0.0) net\_vat: Mapped[float] = mapped\_column(Float, default=0.0) status: Mapped[VATReturnStatus] = mapped\_column(SAEnum(VATReturnStatus), default=VATReturnStatus.DRAFT) created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

python  
from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from sqlalchemy import String, Integer, ForeignKey, DateTime, Enum as SAEnum, Text, Float, UniqueConstraint  
from datetime import datetime  
from .db import Base  
from .enums import InvoiceStatus, Role, OrgStatus  
  
class Organization(Base):  
 \_\_tablename\_\_ = "organizations"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 name: Mapped[str] = mapped\_column(String(255), unique=True, nullable=False)  
 status: Mapped[OrgStatus] = mapped\_column(SAEnum(OrgStatus), default=OrgStatus.APPROVED)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 users: Mapped[list["Membership"]] = relationship(back\_populates="organization")  
 partners: Mapped[list["Partner"]] = relationship(back\_populates="organization")  
 invoices: Mapped[list["Invoice"]] = relationship(back\_populates="organization")  
  
class User(Base):  
 \_\_tablename\_\_ = "users"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 email: Mapped[str] = mapped\_column(String(255), unique=True, index=True)  
 hashed\_password: Mapped[str] = mapped\_column(String(255))  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 memberships: Mapped[list["Membership"]] = relationship(back\_populates="user")  
  
class Membership(Base):  
 \_\_tablename\_\_ = "memberships"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 user\_id: Mapped[int] = mapped\_column(ForeignKey("users.id"))  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"))  
 role: Mapped[Role] = mapped\_column(SAEnum(Role), default=Role.USER)  
  
 user: Mapped[User] = relationship(back\_populates="memberships")  
 organization: Mapped[Organization] = relationship(back\_populates="users")  
  
 \_\_table\_args\_\_ = (UniqueConstraint("user\_id", "org\_id", name="uq\_user\_org"),)  
  
class Partner(Base):  
 \_\_tablename\_\_ = "partners"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 name: Mapped[str] = mapped\_column(String(255), nullable=False)  
 legal\_name: Mapped[str] = mapped\_column(String(255), nullable=True)  
 trn: Mapped[str] = mapped\_column(String(32), nullable=False, index=True)  
 peppol\_participant\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 country\_code: Mapped[str] = mapped\_column(String(2), default="AE", nullable=False)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 organization: Mapped[Organization] = relationship(back\_populates="partners")  
 outbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="seller", foreign\_keys="Invoice.seller\_id")  
 inbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="buyer", foreign\_keys="Invoice.buyer\_id")  
  
class Invoice(Base):  
 \_\_tablename\_\_ = "invoices"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 number: Mapped[str] = mapped\_column(String(64), nullable=False, unique=True)  
  
 seller\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
 buyer\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
  
 issue\_date: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
 due\_days: Mapped[int] = mapped\_column(Integer, default=30)  
  
 currency: Mapped[str] = mapped\_column(String(3), default="AED")  
  
 subtotal: Mapped[float] = mapped\_column(Float, default=0.0)  
 vat\_amount: Mapped[float] = mapped\_column(Float, default=0.0)  
 total: Mapped[float] = mapped\_column(Float, default=0.0)  
  
 status: Mapped[InvoiceStatus] = mapped\_column(SAEnum(InvoiceStatus), default=InvoiceStatus.DRAFT)  
  
 items\_json: Mapped[str] = mapped\_column(Text, default="[]")  
 meta\_json: Mapped[str] = mapped\_column(Text, default="{}")  
  
 provider\_message\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 provider\_status: Mapped[str] = mapped\_column(String(64), nullable=True)  
  
 organization: Mapped[Organization] = relationship(back\_populates="invoices")  
 seller: Mapped[Partner] = relationship(back\_populates="outbound\_invoices", foreign\_keys=[seller\_id])  
 buyer: Mapped[Partner] = relationship(back\_populates="inbound\_invoices", foreign\_keys=[buyer\_id])  
  
# --- Basic Accounting ---  
class AccountType(str):  
 ASSET = "ASSET"  
 LIABILITY = "LIABILITY"  
 EQUITY = "EQUITY"  
 REVENUE = "REVENUE"  
 EXPENSE = "EXPENSE"  
  
class Account(Base):  
 \_\_tablename\_\_ = "accounts"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 code: Mapped[str] = mapped\_column(String(32), index=True)  
 name: Mapped[str] = mapped\_column(String(255))  
 type: Mapped[str] = mapped\_column(String(16))  
  
class JournalEntry(Base):  
 \_\_tablename\_\_ = "journal\_entries"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 ref: Mapped[str] = mapped\_column(String(64))  
 memo: Mapped[str] = mapped\_column(String(255), default="")  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
class JournalLine(Base):  
 \_\_tablename\_\_ = "journal\_lines"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 entry\_id: Mapped[int] = mapped\_column(ForeignKey("journal\_entries.id"), index=True)  
 account\_id: Mapped[int] = mapped\_column(ForeignKey("accounts.id"), index=True)  
 debit: Mapped[float] = mapped\_column(Float, default=0.0)  
 credit: Mapped[float] = mapped\_column(Float, default=0.0)  
  
class Payment(Base):  
 \_\_tablename\_\_ = "payments"  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True)  
 org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
 invoice\_id: Mapped[int] = mapped\_column(ForeignKey("invoices.id"), index=True)  
 amount: Mapped[float] = mapped\_column(Float)  
 method: Mapped[str] = mapped\_column(String(64), default="bank")  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

python from sqlalchemy.orm import Mapped, mapped\_column, relationship from sqlalchemy import String, Integer, ForeignKey, DateTime, Enum as SAEnum, Text, Float, UniqueConstraint from datetime import datetime from .db import Base from .enums import InvoiceStatus, Role

class Organization(Base): **tablename** = “organizations” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) name: Mapped[str] = mapped\_column(String(255), unique=True, nullable=False) created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

users: Mapped[list["Membership"]] = relationship(back\_populates="organization")  
partners: Mapped[list["Partner"]] = relationship(back\_populates="organization")  
invoices: Mapped[list["Invoice"]] = relationship(back\_populates="organization")

class User(Base): **tablename** = “users” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) email: Mapped[str] = mapped\_column(String(255), unique=True, index=True) hashed\_password: Mapped[str] = mapped\_column(String(255)) created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)

memberships: Mapped[list["Membership"]] = relationship(back\_populates="user")

class Membership(Base): **tablename** = “memberships” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) user\_id: Mapped[int] = mapped\_column(ForeignKey(“users.id”)) org\_id: Mapped[int] = mapped\_column(ForeignKey(“organizations.id”)) role: Mapped[Role] = mapped\_column(SAEnum(Role), default=Role.USER)

user: Mapped[User] = relationship(back\_populates="memberships")  
organization: Mapped[Organization] = relationship(back\_populates="users")  
  
\_\_table\_args\_\_ = (UniqueConstraint("user\_id", "org\_id", name="uq\_user\_org"),)

class Partner(Base): **tablename** = “partners”

id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
name: Mapped[str] = mapped\_column(String(255), nullable=False)  
legal\_name: Mapped[str] = mapped\_column(String(255), nullable=True)  
trn: Mapped[str] = mapped\_column(String(32), nullable=False, index=True)  
peppol\_participant\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
country\_code: Mapped[str] = mapped\_column(String(2), default="AE", nullable=False)  
created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
organization: Mapped[Organization] = relationship(back\_populates="partners")  
outbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="seller", foreign\_keys="Invoice.seller\_id")  
inbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="buyer", foreign\_keys="Invoice.buyer\_id")

class Invoice(Base): **tablename** = “invoices”

id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
org\_id: Mapped[int] = mapped\_column(ForeignKey("organizations.id"), index=True)  
number: Mapped[str] = mapped\_column(String(64), nullable=False, unique=True)  
  
seller\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
buyer\_id: Mapped[int] = mapped\_column(ForeKey("partners.id"), nullable=False)  
  
issue\_date: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
due\_days: Mapped[int] = mapped\_column(Integer, default=30)  
  
currency: Mapped[str] = mapped\_column(String(3), default="AED")  
  
subtotal: Mapped[float] = mapped\_column(Float, default=0.0)  
vat\_amount: Mapped[float] = mapped\_column(Float, default=0.0)  
total: Mapped[float] = mapped\_column(Float, default=0.0)  
  
status: Mapped[InvoiceStatus] = mapped\_column(SAEnum(InvoiceStatus), default=InvoiceStatus.DRAFT)  
  
items\_json: Mapped[str] = mapped\_column(Text, default="[]")  
meta\_json: Mapped[str] = mapped\_column(Text, default="{}")  
  
provider\_message\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
provider\_status: Mapped[str] = mapped\_column(String(64), nullable=True)  
  
organization: Mapped[Organization] = relationship(back\_populates="invoices")  
seller: Mapped[Partner] = relationship(back\_populates="outbound\_invoices", foreign\_keys=[seller\_id])  
buyer: Mapped[Partner] = relationship(back\_populates="inbound\_invoices", foreign\_keys=[buyer\_id])

class AuditLog(Base): **tablename** = “audit\_logs” id: Mapped[int] = mapped\_column(Integer, primary\_key=True) timestamp: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow, index=True) user\_id: Mapped[int] = mapped\_column(Integer, index=True) org\_id: Mapped[int] = mapped\_column(Integer, index=True) action: Mapped[str] = mapped\_column(String(64)) resource: Mapped[str] = mapped\_column(String(64)) resource\_id: Mapped[str] = mapped\_column(String(64)) details: Mapped[str] = mapped\_column(Text, default=“{}”)

python  
from sqlalchemy.orm import Mapped, mapped\_column, relationship  
from sqlalchemy import String, Integer, ForeignKey, DateTime, Enum as SAEnum, Text, Float  
from datetime import datetime  
from .db import Base  
from .enums import InvoiceStatus  
  
class Partner(Base):  
 \_\_tablename\_\_ = "partners"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 name: Mapped[str] = mapped\_column(String(255), nullable=False)  
 legal\_name: Mapped[str] = mapped\_column(String(255), nullable=True)  
 trn: Mapped[str] = mapped\_column(String(32), nullable=False, index=True) # UAE VAT TRN (15 digits typical)  
 peppol\_participant\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 country\_code: Mapped[str] = mapped\_column(String(2), default="AE", nullable=False)  
 created\_at: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
  
 outbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="seller", foreign\_keys="Invoice.seller\_id")  
 inbound\_invoices: Mapped[list["Invoice"]] = relationship(back\_populates="buyer", foreign\_keys="Invoice.buyer\_id")  
  
class Invoice(Base):  
 \_\_tablename\_\_ = "invoices"  
  
 id: Mapped[int] = mapped\_column(Integer, primary\_key=True, index=True)  
 number: Mapped[str] = mapped\_column(String(64), nullable=False, unique=True)  
  
 seller\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
 buyer\_id: Mapped[int] = mapped\_column(ForeignKey("partners.id"), nullable=False)  
  
 issue\_date: Mapped[datetime] = mapped\_column(DateTime, default=datetime.utcnow)  
 due\_days: Mapped[int] = mapped\_column(Integer, default=30)  
  
 currency: Mapped[str] = mapped\_column(String(3), default="AED")  
  
 subtotal: Mapped[float] = mapped\_column(Float, default=0.0)  
 vat\_amount: Mapped[float] = mapped\_column(Float, default=0.0)  
 total: Mapped[float] = mapped\_column(Float, default=0.0)  
  
 status: Mapped[InvoiceStatus] = mapped\_column(SAEnum(InvoiceStatus), default=InvoiceStatus.DRAFT)  
  
 # raw JSON of line items and extras for auditability  
 items\_json: Mapped[str] = mapped\_column(Text, default="[]")  
 meta\_json: Mapped[str] = mapped\_column(Text, default="{}")  
  
 # transport fields  
 provider\_message\_id: Mapped[str] = mapped\_column(String(128), nullable=True, index=True)  
 provider\_status: Mapped[str] = mapped\_column(String(64), nullable=True)  
  
 seller: Mapped[Partner] = relationship(back\_populates="outbound\_invoices", foreign\_keys=[seller\_id])  
 buyer: Mapped[Partner] = relationship(back\_populates="inbound\_invoices", foreign\_keys=[buyer\_id])

## app/schemas.py

from pydantic import BaseModel, Field, validator  
from typing import List, Optional, Literal  
from datetime import datetime  
from .enums import InvoiceStatus, Currency, Language  
import re  
  
TRN\_REGEX = re.compile(r"^\d{15}$") # simple UAE TRN check  
  
class PartnerCreate(BaseModel):  
 name: str  
 legal\_name: Optional[str] = None  
 trn: str = Field(..., description="UAE VAT TRN (15 digits)")  
 peppol\_participant\_id: Optional[str] = Field(None, description="PEPPOL participant ID if registered")  
 country\_code: Literal["AE"] = "AE"  
  
 @validator("trn")  
 def validate\_trn(cls, v):  
 if not TRN\_REGEX.match(v):  
 raise ValueError("TRN must be 15 digits")  
 return v  
  
class PartnerOut(PartnerCreate):  
 id: int  
 created\_at: datetime  
  
 class Config:  
 orm\_mode = True  
  
class InvoiceItem(BaseModel):  
 sku: Optional[str] = None  
 description: str  
 quantity: float = 1.0  
 unit\_price: float  
 vat\_rate: float = 0.05 # 5% UAE standard  
  
class InvoiceCreate(BaseModel):  
 number: str  
 seller\_id: int  
 buyer\_id: int  
 issue\_date: Optional[datetime] = None  
 due\_days: int = 30  
 currency: Currency = Currency.AED  
 items: List[InvoiceItem]  
 notes: Optional[str] = None  
  
class InvoiceOut(BaseModel):  
 id: int  
 number: str  
 seller\_id: int  
 buyer\_id: int  
 issue\_date: datetime  
 due\_days: int  
 currency: str  
 subtotal: float  
 vat\_amount: float  
 total: float  
 status: InvoiceStatus  
  
 class Config:  
 orm\_mode = True  
  
class ValidationResult(BaseModel):  
 valid: bool  
 errors: List[str] = []  
  
class SendResult(BaseModel):  
 invoice\_id: int  
 provider\_message\_id: Optional[str]  
 status: str

## app/utils/logging.py

import logging  
import sys  
  
logger = logging.getLogger("beam")  
handler = logging.StreamHandler(sys.stdout)  
formatter = logging.Formatter("%(asctime)s | %(levelname)s | %(name)s | %(message)s")  
handler.setFormatter(formatter)  
logger.addHandler(handler)  
logger.setLevel(logging.INFO)

## app/utils/crypto.py

import base64  
import hashlib  
from typing import Optional  
from cryptography.hazmat.primitives import hashes, serialization  
from cryptography.hazmat.primitives.asymmetric import padding, rsa, ec  
from cryptography.hazmat.primitives.asymmetric.utils import Prehashed  
from cryptography.hazmat.backends import default\_backend  
  
ALG\_MAP = {  
 "sha256": hashes.SHA256,  
}  
  
def sha256(data: bytes) -> str:  
 return hashlib.sha256(data).hexdigest()  
  
def load\_private\_key(pem: str):  
 return serialization.load\_pem\_private\_key(pem.encode(), password=None, backend=default\_backend())  
  
def sign\_detached(data: bytes, private\_key\_pem: Optional[str], hash\_alg: str = "sha256") -> Optional[str]:  
 if not private\_key\_pem:  
 return None  
 key = load\_private\_key(private\_key\_pem)  
 h = ALG\_MAP.get(hash\_alg, hashes.SHA256)()  
 if isinstance(key, rsa.RSAPrivateKey):  
 sig = key.sign(data, padding.PKCS1v15(), h)  
 elif isinstance(key, ec.EllipticCurvePrivateKey):  
 sig = key.sign(data, ec.ECDSA(h))  
 else:  
 raise ValueError("Unsupported key type")  
 return base64.b64encode(sig).decode()

python # Placeholder for future signing/encryption (UAE requirements, PEPPOL AS4 if needed) # Keep interfaces simple for now. from typing import Optional

def sign\_payload(data: bytes, private\_key\_pem: Optional[str] = None) -> bytes: # TODO: implement proper signing return data

## app/middleware/request\_context.py  
```python  
from starlette.middleware.base import BaseHTTPMiddleware  
from starlette.requests import Request  
import uuid  
  
class RequestContextMiddleware(BaseHTTPMiddleware):  
 async def dispatch(self, request: Request, call\_next):  
 request.state.request\_id = str(uuid.uuid4())  
 response = await call\_next(request)  
 response.headers["X-Request-ID"] = request.state.request\_id  
 return response

## app/auth.py

from datetime import datetime, timedelta  
from typing import Optional  
from fastapi import Header, HTTPException, status, Depends, APIRouter, Form  
from fastapi.security import OAuth2PasswordBearer  
from jose import jwt, JWTError  
from passlib.context import CryptContext  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select  
from .config import settings  
from .models import User, Organization, Membership  
from .enums import Role, OrgStatus  
from .db import get\_session  
  
pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")  
oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/api/v1/auth/token")  
  
def verify\_password(plain\_password, hashed\_password):  
 return pwd\_context.verify(plain\_password, hashed\_password)  
  
def get\_password\_hash(password):  
 return pwd\_context.hash(password)  
  
def create\_access\_token(data: dict, expires\_delta: Optional[timedelta] = None):  
 to\_encode = data.copy()  
 expire = datetime.utcnow() + (expires\_delta or timedelta(minutes=settings.ACCESS\_TOKEN\_EXPIRE\_MINUTES))  
 to\_encode.update({"exp": expire})  
 encoded\_jwt = jwt.encode(to\_encode, settings.JWT\_SECRET, algorithm=settings.JWT\_ALGORITHM)  
 return encoded\_jwt  
  
async def api\_key\_auth(x\_api\_key: str = Header(None)):  
 if x\_api\_key != settings.API\_KEY:  
 raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid API key")  
 return True  
  
async def get\_current\_user(db: AsyncSession = Depends(get\_session), token: str = Depends(oauth2\_scheme)) -> User:  
 credentials\_exception = HTTPException(status\_code=401, detail="Could not validate credentials")  
 try:  
 payload = jwt.decode(token, settings.JWT\_SECRET, algorithms=[settings.JWT\_ALGORITHM])  
 user\_id: int = payload.get("sub")  
 if user\_id is None:  
 raise credentials\_exception  
 except JWTError:  
 raise credentials\_exception  
 res = await db.execute(select(User).where(User.id == int(user\_id)))  
 user = res.scalar\_one\_or\_none()  
 if not user:  
 raise credentials\_exception  
 return user  
  
async def require\_membership(org\_id: int, user: User, db: AsyncSession) -> Membership:  
 res = await db.execute(select(Membership).where(Membership.user\_id == user.id, Membership.org\_id == org\_id))  
 m = res.scalar\_one\_or\_none()  
 if not m:  
 raise HTTPException(status\_code=403, detail="Not a member of this organization")  
 return m  
  
async def require\_superadmin(user: User, db: AsyncSession) -> None:  
 # superadmin = membership with role SUPERADMIN on a special org (id=0) OR email whitelist  
 res = await db.execute(select(Membership).where(Membership.user\_id == user.id, Membership.role == Role.SUPERADMIN))  
 if not res.scalars().first():  
 raise HTTPException(status\_code=403, detail="Requires SUPERADMIN")  
  
router = APIRouter(prefix=f"{settings.API\_V1\_PREFIX}/auth", tags=["auth"])  
  
@router.post("/signup")  
async def signup(email: str = Form(...), password: str = Form(...), org\_name: str = Form(...), db: AsyncSession = Depends(get\_session)):  
 existing\_user = (await db.execute(select(User).where(User.email == email))).scalar\_one\_or\_none()  
 if existing\_user:  
 raise HTTPException(status\_code=400, detail="User already exists")  
 org = Organization(name=org\_name, status=(OrgStatus.APPROVED if settings.AUTO\_APPROVE\_SIGNUPS else OrgStatus.PENDING))  
 user = User(email=email, hashed\_password=get\_password\_hash(password))  
 db.add\_all([org, user])  
 await db.flush()  
 m = Membership(user\_id=user.id, org\_id=org.id, role=Role.OWNER)  
 db.add(m)  
 await db.commit()  
 token = create\_access\_token({"sub": user.id, "org": org.id})  
 return {"access\_token": token, "token\_type": "bearer", "org\_id": org.id, "org\_status": org.status.value}  
  
@router.post("/token")  
async def login\_for\_access\_token(email: str = Form(...), password: str = Form(...), db: AsyncSession = Depends(get\_session)):  
 res = await db.execute(select(User).where(User.email == email))  
 user = res.scalar\_one\_or\_none()  
 if not user or not verify\_password(password, user.hashed\_password):  
 raise HTTPException(status\_code=400, detail="Incorrect email or password")  
 mem = (await db.execute(select(Membership).where(Membership.user\_id == user.id))).scalars().first()  
 org\_id = mem.org\_id if mem else None  
 token = create\_access\_token({"sub": user.id, "org": org\_id})  
 return {"access\_token": token, "token\_type": "bearer", "org\_id": org\_id}

python from fastapi import Header, HTTPException, status, Depends from .config import settings

async def api\_key\_auth(x\_api\_key: str = Header(None)): if x\_api\_key != settings.API\_KEY: raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail=“Invalid API key”) return True

## app/peppol/validators.py  
```python  
from typing import List  
from ..schemas import InvoiceCreate  
  
REQUIRED\_SELLER\_FIELDS = ["seller\_id"]  
REQUIRED\_BUYER\_FIELDS = ["buyer\_id"]  
  
# Very small subset of rules (expand as needed with official specs)  
def validate\_invoice\_business\_rules(payload: InvoiceCreate) -> List[str]:  
 errors: List[str] = []  
 if not payload.items or len(payload.items) == 0:  
 errors.append("At least one invoice item is required")  
 if payload.due\_days < 0:  
 errors.append("due\_days cannot be negative")  
 # Check values  
 for idx, item in enumerate(payload.items):  
 if item.quantity <= 0:  
 errors.append(f"Item {idx+1}: quantity must be > 0")  
 if item.unit\_price < 0:  
 errors.append(f"Item {idx+1}: unit\_price must be >= 0")  
 if not (0.0 <= item.vat\_rate <= 1.0):  
 errors.append(f"Item {idx+1}: vat\_rate must be between 0 and 1")  
 return errors

## app/peppol/xml\_builder.py

from lxml import etree  
from datetime import datetime, timedelta  
from ..schemas import InvoiceCreate  
from ..config import settings  
  
# Namespaces for UBL 2.1 Invoice  
NSMAP = {  
 None: "urn:oasis:names:specification:ubl:schema:xsd:Invoice-2",  
 "cbc": "urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2",  
 "cac": "urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2",  
}  
  
# Minimal UBL 2.1-conformant structure for MVP. For production, include full mandatory elements.  
  
def build\_invoice\_xml(payload: InvoiceCreate, seller\_trn: str, buyer\_trn: str) -> bytes:  
 issue\_date = payload.issue\_date or datetime.utcnow()  
 due\_date = issue\_date + timedelta(days=payload.due\_days)  
  
 inv = etree.Element("Invoice", nsmap=NSMAP)  
  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}CustomizationID").text = "urn:fdc:peppol.eu:poacc:billing:3"  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}ProfileID").text = "urn:fdc:peppol.eu:poacc:billing:3:ver3.0"  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}ID").text = payload.number  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}IssueDate").text = issue\_date.strftime("%Y-%m-%d")  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}DueDate").text = due\_date.strftime("%Y-%m-%d")  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}DocumentCurrencyCode").text = payload.currency.value  
  
 # Supplier  
 asp = etree.SubElement(inv, f"{{{NSMAP['cac']}}}AccountingSupplierParty")  
 party = etree.SubElement(asp, f"{{{NSMAP['cac']}}}Party")  
 pid = etree.SubElement(party, f"{{{NSMAP['cac']}}}PartyIdentification")  
 etree.SubElement(pid, f"{{{NSMAP['cbc']}}}ID").text = seller\_trn  
  
 # Customer  
 acp = etree.SubElement(inv, f"{{{NSMAP['cac']}}}AccountingCustomerParty")  
 party2 = etree.SubElement(acp, f"{{{NSMAP['cac']}}}Party")  
 pid2 = etree.SubElement(party2, f"{{{NSMAP['cac']}}}PartyIdentification")  
 etree.SubElement(pid2, f"{{{NSMAP['cbc']}}}ID").text = buyer\_trn  
  
 # Lines & totals  
 subtotal = 0.0  
 vat\_total = 0.0  
 for idx, item in enumerate(payload.items, start=1):  
 il = etree.SubElement(inv, f"{{{NSMAP['cac']}}}InvoiceLine")  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}ID").text = str(idx)  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}InvoicedQuantity").text = str(item.quantity)  
 line\_amount = item.quantity \* item.unit\_price  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}LineExtensionAmount", currencyID=payload.currency.value).text = f"{line\_amount:.2f}"  
  
 item\_node = etree.SubElement(il, f"{{{NSMAP['cac']}}}Item")  
 etree.SubElement(item\_node, f"{{{NSMAP['cbc']}}}Description").text = item.description  
  
 price = etree.SubElement(il, f"{{{NSMAP['cac']}}}Price")  
 etree.SubElement(price, f"{{{NSMAP['cbc']}}}PriceAmount", currencyID=payload.currency.value).text = f"{item.unit\_price:.2f}"  
  
 subtotal += line\_amount  
 vat\_total += line\_amount \* item.vat\_rate  
  
 lmt = etree.SubElement(inv, f"{{{NSMAP['cac']}}}LegalMonetaryTotal")  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}LineExtensionAmount", currencyID=payload.currency.value).text = f"{subtotal:.2f}"  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}TaxExclusiveAmount", currencyID=payload.currency.value).text = f"{subtotal:.2f}"  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}TaxAmount", currencyID=payload.currency.value).text = f"{vat\_total:.2f}"  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}PayableAmount", currencyID=payload.currency.value).text = f"{subtotal + vat\_total:.2f}"  
  
 xml\_bytes = etree.tostring(inv, xml\_declaration=True, encoding="UTF-8")  
  
 # Optional XSD validation  
 if settings.UBL\_XSD\_PATH:  
 with open(settings.UBL\_XSD\_PATH, "rb") as f:  
 schema\_doc = etree.parse(f)  
 schema = etree.XMLSchema(schema\_doc)  
 schema.assertValid(etree.fromstring(xml\_bytes))  
  
 return xml\_bytes

python from xml.etree.ElementTree import Element, SubElement, tostring from datetime import datetime, timedelta from ..schemas import InvoiceCreate

# NOTE: This is a simplified PEPPOL BIS 3.0-like builder for MVP demo/testing.

# In production, use official UBL 2.1 schema, namespaces, and XSD validation.

def build\_invoice\_xml(payload: InvoiceCreate, seller\_trn: str, buyer\_trn: str) -> bytes: root = Element(“Invoice”) SubElement(root, “CustomizationID”).text = “PEPPOL-BIS3-MVP” SubElement(root, “ProfileID”).text = “urn:fdc:peppol.eu:poacc:billing:3”

SubElement(root, "ID").text = payload.number  
issue\_date = payload.issue\_date or datetime.utcnow()  
SubElement(root, "IssueDate").text = issue\_date.strftime("%Y-%m-%d")  
SubElement(root, "DueDate").text = (issue\_date + timedelta(days=payload.due\_days)).strftime("%Y-%m-%d")  
SubElement(root, "DocumentCurrencyCode").text = payload.currency.value  
  
acct = SubElement(root, "AccountingSupplierParty")  
SubElement(acct, "CompanyID").text = seller\_trn  
  
cust = SubElement(root, "AccountingCustomerParty")  
SubElement(cust, "CompanyID").text = buyer\_trn  
  
# Lines  
lines = SubElement(root, "InvoiceLines")  
subtotal = 0.0  
vat\_total = 0.0  
for idx, item in enumerate(payload.items, start=1):  
 line = SubElement(lines, "InvoiceLine")  
 SubElement(line, "ID").text = str(idx)  
 SubElement(line, "InvoicedQuantity").text = str(item.quantity)  
 SubElement(line, "LineExtensionAmount").text = f"{item.quantity \* item.unit\_price:.2f}"  
 SubElement(line, "Description").text = item.description  
 SubElement(line, "VATRate").text = f"{item.vat\_rate:.2f}"  
 subtotal += item.quantity \* item.unit\_price  
 vat\_total += item.quantity \* item.unit\_price \* item.vat\_rate  
  
# Totals  
legal = SubElement(root, "LegalMonetaryTotal")  
SubElement(legal, "LineExtensionAmount").text = f"{subtotal:.2f}"  
SubElement(legal, "TaxExclusiveAmount").text = f"{subtotal:.2f}"  
SubElement(legal, "TaxAmount").text = f"{vat\_total:.2f}"  
SubElement(legal, "PayableAmount").text = f"{subtotal + vat\_total:.2f}"  
  
return tostring(root)

## app/peppol/provider\_client.py  
```python  
from typing import Optional  
import httpx  
from ..config import settings  
from ..utils.logging import logger  
  
class ProviderClient:  
 def \_\_init\_\_(self, base\_url: Optional[str] = None, api\_key: Optional[str] = None, timeout: int = 20):  
 self.base\_url = base\_url or settings.PROVIDER\_BASE\_URL  
 self.api\_key = api\_key or settings.PROVIDER\_API\_KEY  
 self.timeout = timeout or settings.PROVIDER\_TIMEOUT\_SECONDS  
  
 async def send\_invoice(self, xml\_payload: bytes) -> dict:  
 """Send invoice to accredited PEPPOL provider API.  
 If PROVIDER\_BASE\_URL is not configured, we simulate a successful send.  
 """  
 if not self.base\_url:  
 logger.info("Provider base URL not set; simulating send.")  
 return {"message\_id": "SIM-12345", "status": "ACCEPTED", "simulated": True}  
  
 headers = {"Authorization": f"Bearer {self.api\_key}", "Content-Type": "application/xml"}  
 async with httpx.AsyncClient(timeout=self.timeout) as client:  
 resp = await client.post(f"{self.base\_url}/invoices", content=xml\_payload, headers=headers)  
 resp.raise\_for\_status()  
 data = resp.json()  
 logger.info("Provider response: %s", data)  
 return data

## app/services/partner\_service.py

from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select  
from ..models import Partner  
from ..schemas import PartnerCreate  
  
class PartnerService:  
 @staticmethod  
 async def create(db: AsyncSession, payload: PartnerCreate, \*, org\_id: int) -> Partner:  
 existing = await db.execute(select(Partner).where(Partner.trn == payload.trn, Partner.org\_id == org\_id))  
 if existing.scalar\_one\_or\_none():  
 raise ValueError("Partner with this TRN already exists in this org")  
 partner = Partner(  
 org\_id=org\_id,  
 name=payload.name,  
 legal\_name=payload.legal\_name,  
 trn=payload.trn,  
 peppol\_participant\_id=payload.peppol\_participant\_id,  
 country\_code=payload.country\_code,  
 )  
 db.add(partner)  
 await db.commit()  
 await db.refresh(partner)  
 return partner  
  
 @staticmethod  
 async def list\_all(db: AsyncSession, \*, org\_id: int) -> list[Partner]:  
 res = await db.execute(select(Partner).where(Partner.org\_id == org\_id).order\_by(Partner.id.desc()))  
 return res.scalars().all()  
  
 @staticmethod  
 async def get\_by\_id(db: AsyncSession, partner\_id: int, \*, org\_id: int) -> Partner | None:  
 res = await db.execute(select(Partner).where(Partner.id == partner\_id, Partner.org\_id == org\_id))  
 return res.scalar\_one\_or\_none()

python from sqlalchemy.ext.asyncio import AsyncSession from sqlalchemy import select from ..models import Partner from ..schemas import PartnerCreate

class PartnerService: @staticmethod async def create(db: AsyncSession, payload: PartnerCreate) -> Partner: existing = await db.execute(select(Partner).where(Partner.trn == payload.trn)) if existing.scalar\_one\_or\_none(): raise ValueError(“Partner with this TRN already exists”) partner = Partner( name=payload.name, legal\_name=payload.legal\_name, trn=payload.trn, peppol\_participant\_id=payload.peppol\_participant\_id, country\_code=payload.country\_code, ) db.add(partner) await db.commit() await db.refresh(partner) return partner

@staticmethod  
async def list\_all(db: AsyncSession) -> list[Partner]:  
 res = await db.execute(select(Partner).order\_by(Partner.id.desc()))  
 return res.scalars().all()  
  
@staticmethod  
async def get\_by\_id(db: AsyncSession, partner\_id: int) -> Partner | None:  
 res = await db.execute(select(Partner).where(Partner.id == partner\_id))  
 return res.scalar\_one\_or\_none()

## app/services/invoice\_service.py  
```python  
import json  
from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select, and\_  
from ..models import Invoice, Partner, Account, AccountType, JournalEntry, JournalLine  
from ..schemas import InvoiceCreate, ValidationResult  
from ..enums import InvoiceStatus  
from ..peppol.validators import validate\_invoice\_business\_rules  
from ..peppol.xml\_builder import build\_invoice\_xml  
from ..peppol.provider\_client import ProviderClient  
from ..config import settings  
from ..utils.crypto import sha256, sign\_detached  
  
async def ensure\_core\_accounts(db: AsyncSession, org\_id: int) -> dict[str, Account]:  
 async def get\_or\_create(code: str, name: str, typ: str) -> Account:  
 res = await db.execute(select(Account).where(Account.org\_id == org\_id, Account.code == code))  
 acc = res.scalar\_one\_or\_none()  
 if not acc:  
 acc = Account(org\_id=org\_id, code=code, name=name, type=typ)  
 db.add(acc)  
 await db.flush()  
 return acc  
 ar = await get\_or\_create("1100", "Accounts Receivable", AccountType.ASSET)  
 ap = await get\_or\_create("2100", "Accounts Payable", AccountType.LIABILITY)  
 rev = await get\_or\_create("4000", "Sales Revenue", AccountType.REVENUE)  
 vat\_out = await get\_or\_create("2200", "VAT Payable (Output)", AccountType.LIABILITY)  
 vat\_in = await get\_or\_create("1300", "VAT Receivable (Input)", AccountType.ASSET)  
 cash = await get\_or\_create("1000", "Cash", AccountType.ASSET)  
 exp = await get\_or\_create("5000", "Expenses", AccountType.EXPENSE)  
 await db.commit()  
 return {"AR": ar, "AP": ap, "REV": rev, "VAT\_OUT": vat\_out, "VAT\_IN": vat\_in, "CASH": cash, "EXP": exp}  
  
class InvoiceService:  
 @staticmethod  
 async def create(db: AsyncSession, payload: InvoiceCreate, \*, org\_id: int) -> Invoice:  
 subtotal = sum([i.quantity \* i.unit\_price for i in payload.items])  
 vat\_amount = sum([i.quantity \* i.unit\_price \* i.vat\_rate for i in payload.items])  
 sign\_factor = -1.0 if payload.invoice\_type in {"CREDIT\_NOTE\_SALE", "CREDIT\_NOTE\_PURCHASE"} else 1.0  
 subtotal \*= sign\_factor  
 vat\_amount \*= sign\_factor  
 total = subtotal + vat\_amount  
  
 invoice = Invoice(  
 org\_id=org\_id,  
 number=payload.number,  
 invoice\_type=payload.invoice\_type,  
 seller\_id=payload.seller\_id,  
 buyer\_id=payload.buyer\_id,  
 issue\_date=payload.issue\_date or datetime.utcnow(),  
 due\_days=payload.due\_days,  
 currency=payload.currency.value,  
 subtotal=subtotal,  
 vat\_amount=vat\_amount,  
 total=total,  
 status=InvoiceStatus.DRAFT,  
 items\_json=json.dumps([i.dict() for i in payload.items]),  
 meta\_json=json.dumps({"notes": payload.notes or ""}),  
 )  
 db.add(invoice)  
 await db.flush()  
  
 # --- Hash chain (based on canonical XML bytes) ---  
 # Find previous hashed invoice for this org by issue\_date then id  
 prev = (await db.execute(select(Invoice).where(and\_(Invoice.org\_id==org\_id, Invoice.curr\_hash!=None)).order\_by(Invoice.issue\_date.desc(), Invoice.id.desc()))).scalars().first()  
 prev\_hash = prev.curr\_hash if prev else None  
 # Build canonical XML for hash  
 # Note: we need seller/buyer TRNs for XML  
 seller = await db.get(Partner, invoice.seller\_id)  
 buyer = await db.get(Partner, invoice.buyer\_id)  
 payload\_for\_xml = InvoiceCreate(  
 number=invoice.number,  
 seller\_id=invoice.seller\_id,  
 buyer\_id=invoice.buyer\_id,  
 invoice\_type=invoice.invoice\_type,  
 issue\_date=invoice.issue\_date,  
 due\_days=invoice.due\_days,  
 items=json.loads(invoice.items\_json),  
 currency=invoice.currency,  
 )  
 xml\_bytes = build\_invoice\_xml(payload\_for\_xml, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 # Mix prev hash into digest base (chain)  
 to\_hash = (prev\_hash or "").encode() + xml\_bytes  
 curr\_hash = sha256(to\_hash)  
 signature\_b64 = sign\_detached(to\_hash, settings.SIGNING\_PRIVATE\_KEY\_PEM, settings.HASH\_ALG)  
  
 invoice.prev\_hash = prev\_hash  
 invoice.curr\_hash = curr\_hash  
 invoice.signature\_b64 = signature\_b64  
 invoice.signing\_cert\_serial = settings.SIGNING\_CERT\_SERIAL  
  
 await db.commit()  
 await db.refresh(invoice)  
  
 # Accounting posts  
 accts = await ensure\_core\_accounts(db, org\_id)  
 je = JournalEntry(org\_id=org\_id, ref=invoice.number, memo=f"Invoice {payload.invoice\_type}")  
 db.add(je)  
 await db.flush()  
  
 if payload.invoice\_type in {"SALE", "CREDIT\_NOTE\_SALE"}:  
 db.add\_all([  
 JournalLine(entry\_id=je.id, account\_id=accts["AR"].id, debit=total if total>0 else 0.0, credit=abs(total) if total<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["REV"].id, debit=abs(subtotal) if subtotal<0 else 0.0, credit=subtotal if subtotal>0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["VAT\_OUT"].id, debit=abs(vat\_amount) if vat\_amount<0 else 0.0, credit=vat\_amount if vat\_amount>0 else 0.0),  
 ])  
 else:  
 expense\_amount = subtotal  
 db.add\_all([  
 JournalLine(entry\_id=je.id, account\_id=accts["EXP"].id, debit=expense\_amount if expense\_amount>0 else 0.0, credit=abs(expense\_amount) if expense\_amount<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["VAT\_IN"].id, debit=vat\_amount if vat\_amount>0 else 0.0, credit=abs(vat\_amount) if vat\_amount<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["AP"].id, debit=abs(total) if total<0 else 0.0, credit=total if total>0 else 0.0),  
 ])  
  
 await db.commit()  
 return invoice  
  
 @staticmethod  
 async def get(db: AsyncSession, invoice\_id: int, \*, org\_id: int) -> Invoice | None:  
 res = await db.execute(select(Invoice).where(Invoice.id == invoice\_id, Invoice.org\_id == org\_id))  
 return res.scalar\_one\_or\_none()  
  
 @staticmethod  
 async def validate(db: AsyncSession, payload: InvoiceCreate, \*, org\_id: int) -> ValidationResult:  
 errors = validate\_invoice\_business\_rules(payload)  
 seller = await db.get(Partner, payload.seller\_id)  
 buyer = await db.get(Partner, payload.buyer\_id)  
 if not seller or seller.org\_id != org\_id:  
 errors.append("Seller not found in org")  
 if not buyer or buyer.org\_id != org\_id:  
 errors.append("Buyer not found in org")  
 return ValidationResult(valid=len(errors) == 0, errors=errors)  
  
 @staticmethod  
 async def send(db: AsyncSession, invoice\_id: int, provider: ProviderClient, \*, org\_id: int) -> Invoice:  
 invoice = await db.get(Invoice, invoice\_id)  
 if not invoice or invoice.org\_id != org\_id:  
 raise ValueError("Invoice not found in org")  
 seller = await db.get(Partner, invoice.seller\_id)  
 buyer = await db.get(Partner, invoice.buyer\_id)  
 payload = InvoiceCreate(  
 number=invoice.number,  
 seller\_id=invoice.seller\_id,  
 buyer\_id=invoice.buyer\_id,  
 invoice\_type=invoice.invoice\_type,  
 issue\_date=invoice.issue\_date,  
 due\_days=invoice.due\_days,  
 items=json.loads(invoice.items\_json),  
 currency=invoice.currency,  
 )  
 xml = build\_invoice\_xml(payload, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 resp = await provider.send\_invoice(xml)  
 invoice.provider\_message\_id = resp.get("message\_id")  
 invoice.provider\_status = resp.get("status")  
 invoice.status = InvoiceStatus.SENT if resp.get("status") in {"ACCEPTED", "PENDING"} else InvoiceStatus.REJECTED  
 await db.commit()  
 await db.refresh(invoice)  
 return invoice

@staticmethod  
async def get(db: AsyncSession, invoice\_id: int, \*, org\_id: int) -> Invoice | None:  
 res = await db.execute(select(Invoice).where(Invoice.id == invoice\_id, Invoice.org\_id == org\_id))  
 return res.scalar\_one\_or\_none()  
  
@staticmethod  
async def validate(db: AsyncSession, payload: InvoiceCreate, \*, org\_id: int) -> ValidationResult:  
 errors = validate\_invoice\_business\_rules(payload)  
 seller = await db.get(Partner, payload.seller\_id)  
 buyer = await db.get(Partner, payload.buyer\_id)  
 if not seller or seller.org\_id != org\_id:  
 errors.append("Seller not found in org")  
 if not buyer or buyer.org\_id != org\_id:  
 errors.append("Buyer not found in org")  
 return ValidationResult(valid=len(errors) == 0, errors=errors)  
  
@staticmethod  
async def send(db: AsyncSession, invoice\_id: int, provider: ProviderClient, \*, org\_id: int) -> Invoice:  
 invoice = await db.get(Invoice, invoice\_id)  
 if not invoice or invoice.org\_id != org\_id:  
 raise ValueError("Invoice not found in org")  
 seller = await db.get(Partner, invoice.seller\_id)  
 buyer = await db.get(Partner, invoice.buyer\_id)  
 payload = InvoiceCreate(  
 number=invoice.number,  
 seller\_id=invoice.seller\_id,  
 buyer\_id=invoice.buyer\_id,  
 issue\_date=invoice.issue\_date,  
 due\_days=invoice.due\_days,  
 items=json.loads(invoice.items\_json),  
 currency=invoice.currency,  
 )  
 xml = build\_invoice\_xml(payload, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 resp = await provider.send\_invoice(xml)  
 invoice.provider\_message\_id = resp.get("message\_id")  
 invoice.provider\_status = resp.get("status")  
 invoice.status = InvoiceStatus.SENT if resp.get("status") in {"ACCEPTED", "PENDING"} else InvoiceStatus.REJECTED  
 await db.commit()  
 await db.refresh(invoice)  
 return invoice

python  
import json  
from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select  
from ..models import Invoice, Partner  
from ..schemas import InvoiceCreate, ValidationResult  
from ..enums import InvoiceStatus  
from ..peppol.validators import validate\_invoice\_business\_rules  
from ..peppol.xml\_builder import build\_invoice\_xml  
from ..peppol.provider\_client import ProviderClient  
  
class InvoiceService:  
 @staticmethod  
 async def create(db: AsyncSession, payload: InvoiceCreate) -> Invoice:  
 # compute totals  
 subtotal = sum([i.quantity \* i.unit\_price for i in payload.items])  
 vat\_amount = sum([i.quantity \* i.unit\_price \* i.vat\_rate for i in payload.items])  
 total = subtotal + vat\_amount  
  
 invoice = Invoice(  
 number=payload.number,  
 seller\_id=payload.seller\_id,  
 buyer\_id=payload.buyer\_id,  
 issue\_date=payload.issue\_date or datetime.utcnow(),  
 due\_days=payload.due\_days,  
 currency=payload.currency.value,  
 subtotal=subtotal,  
 vat\_amount=vat\_amount,  
 total=total,  
 status=InvoiceStatus.DRAFT,  
 items\_json=json.dumps([i.dict() for i in payload.items]),  
 meta\_json=json.dumps({"notes": payload.notes or ""}),  
 )  
 db.add(invoice)  
 await db.commit()  
 await db.refresh(invoice)  
 return invoice  
  
 @staticmethod  
 async def get(db: AsyncSession, invoice\_id: int) -> Invoice | None:  
 res = await db.execute(select(Invoice).where(Invoice.id == invoice\_id))  
 return res.scalar\_one\_or\_none()  
  
 @staticmethod  
 async def validate(db: AsyncSession, payload: InvoiceCreate) -> ValidationResult:  
 errors = validate\_invoice\_business\_rules(payload)  
 # existence checks  
 seller = await db.get(Partner, payload.seller\_id)  
 buyer = await db.get(Partner, payload.buyer\_id)  
 if not seller:  
 errors.append("Seller not found")  
 if not buyer:  
 errors.append("Buyer not found")  
 return ValidationResult(valid=len(errors) == 0, errors=errors)  
  
 @staticmethod  
 async def send(db: AsyncSession, invoice\_id: int, provider: ProviderClient) -> Invoice:  
 invoice = await db.get(Invoice, invoice\_id)  
 if not invoice:  
 raise ValueError("Invoice not found")  
 seller = await db.get(Partner, invoice.seller\_id)  
 buyer = await db.get(Partner, invoice.buyer\_id)  
 # reconstruct payload for XML  
 payload = InvoiceCreate(  
 number=invoice.number,  
 seller\_id=invoice.seller\_id,  
 buyer\_id=invoice.buyer\_id,  
 issue\_date=invoice.issue\_date,  
 due\_days=invoice.due\_days,  
 items=json.loads(invoice.items\_json),  
 currency=invoice.currency,  
 )  
 xml = build\_invoice\_xml(payload, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 resp = await provider.send\_invoice(xml)  
 invoice.provider\_message\_id = resp.get("message\_id")  
 invoice.provider\_status = resp.get("status")  
 invoice.status = InvoiceStatus.SENT if resp.get("status") in {"ACCEPTED", "PENDING"} else InvoiceStatus.REJECTED  
 await db.commit()  
 await db.refresh(invoice)  
 return invoice

## app/main.py

import uvicorn  
from fastapi import FastAPI, Depends, HTTPException, Header, Query  
from fastapi.responses import StreamingResponse, JSONResponse  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select  
from datetime import datetime  
from .config import settings  
from .db import Base, engine, get\_session  
from .auth import get\_current\_user, require\_membership, require\_superadmin, router as auth\_router  
from .schemas import PartnerCreate, PartnerOut, InvoiceCreate, InvoiceOut, ValidationResult  
from .services.partner\_service import PartnerService  
from .services.invoice\_service import InvoiceService  
from .peppol.provider\_client import ProviderClient  
from .middleware.request\_context import RequestContextMiddleware  
from .utils.audit import log\_audit  
from .models import Organization, Invoice  
from .enums import OrgStatus  
from .reports.audit\_export import build\_audit\_zip  
from .accounting.financials import trial\_balance, income\_statement  
from .vat.returns import prepare\_vat\_return  
from .vat.form201 import preview\_form201  
  
app = FastAPI(title=settings.APP\_NAME)  
app.add\_middleware(RequestContextMiddleware)  
app.include\_router(auth\_router)  
  
@app.on\_event("startup")  
async def startup():  
 async with engine.begin() as conn:  
 await conn.run\_sync(Base.metadata.create\_all)  
  
# Helpers  
async def org\_guard(org\_id: int = Header(..., alias="X-Org-ID"), user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_membership(org\_id, user, db)  
 org = (await db.execute(select(Organization).where(Organization.id == org\_id))).scalar\_one()  
 if org.status != OrgStatus.APPROVED:  
 raise HTTPException(status\_code=403, detail="Organization not approved yet")  
 return org\_id, user  
  
# Admin endpoints  
@app.get(f"{settings.API\_V1\_PREFIX}/admin/orgs/pending")  
async def list\_pending\_orgs(user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_superadmin(user, db)  
 rows = (await db.execute(select(Organization).where(Organization.status == OrgStatus.PENDING))).scalars().all()  
 return [{"id": o.id, "name": o.name, "status": o.status.value} for o in rows]  
  
@app.post(f"{settings.API\_V1\_PREFIX}/admin/orgs/{{org\_id}}/approve")  
async def approve\_org(org\_id: int, user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_superadmin(user, db)  
 org = (await db.execute(select(Organization).where(Organization.id == org\_id))).scalar\_one\_or\_none()  
 if not org:  
 raise HTTPException(status\_code=404, detail="Org not found")  
 org.status = OrgStatus.APPROVED  
 await db.commit()  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="APPROVE", resource="Organization", resource\_id=org\_id, details={})  
 return {"id": org.id, "status": org.status.value}  
  
# Partners  
@app.post(f"{settings.API\_V1\_PREFIX}/partners", response\_model=PartnerOut)  
async def create\_partner(payload: PartnerCreate, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 try:  
 partner = await PartnerService.create(db, payload, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="CREATE", resource="Partner", resource\_id=partner.id, details=payload.dict())  
 return partner  
 except ValueError as e:  
 raise HTTPException(status\_code=400, detail=str(e))  
  
@app.get(f"{settings.API\_V1\_PREFIX}/partners", response\_model=list[PartnerOut])  
async def list\_partners(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 items = await PartnerService.list\_all(db, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="LIST", resource="Partner", resource\_id="\*", details={})  
 return items  
  
# Invoices  
@app.post(f"{settings.API\_V1\_PREFIX}/invoices", response\_model=InvoiceOut)  
async def create\_invoice(payload: InvoiceCreate, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 vr = await InvoiceService.validate(db, payload, org\_id=org\_id)  
 if not vr.valid:  
 raise HTTPException(status\_code=400, detail=vr.errors)  
 inv = await InvoiceService.create(db, payload, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="CREATE", resource="Invoice", resource\_id=inv.id, details=payload.dict())  
 return inv  
  
@app.get(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}", response\_model=InvoiceOut)  
async def get\_invoice(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 inv = await InvoiceService.get(db, invoice\_id, org\_id=org\_id)  
 if not inv:  
 raise HTTPException(status\_code=404, detail="Invoice not found")  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="READ", resource="Invoice", resource\_id=invoice\_id, details={})  
 return inv  
  
@app.get(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}/signature")  
async def get\_invoice\_signature(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 inv = await InvoiceService.get(db, invoice\_id, org\_id=org\_id)  
 if not inv:  
 raise HTTPException(status\_code=404, detail="Invoice not found")  
 return {"prev\_hash": inv.prev\_hash, "curr\_hash": inv.curr\_hash, "signature\_b64": inv.signature\_b64, "signing\_cert\_serial": inv.signing\_cert\_serial}  
  
@app.post(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}/send", response\_model=InvoiceOut)  
async def send\_invoice(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 provider = ProviderClient()  
 try:  
 inv = await InvoiceService.send(db, invoice\_id, provider, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="SEND", resource="Invoice", resource\_id=invoice\_id, details={"provider\_status": inv.provider\_status})  
 return inv  
 except ValueError as e:  
 raise HTTPException(status\_code=404, detail=str(e))  
  
# ---- Compliance & Reporting ----  
@app.get(f"{settings.API\_V1\_PREFIX}/compliance/audit-file")  
async def export\_audit\_file(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session), start: str = Query(...), end: str = Query(...)):  
 org\_id, user = deps  
 start\_dt = datetime.fromisoformat(start)  
 end\_dt = datetime.fromisoformat(end)  
 zip\_bytes, filename = await build\_audit\_zip(db, org\_id=org\_id, start=start\_dt, end=end\_dt)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="EXPORT", resource="AuditFile", resource\_id="\*", details={"start": start, "end": end})  
 return StreamingResponse(iter([zip\_bytes]), media\_type="application/zip", headers={"Content-Disposition": f"attachment; filename={filename}"})  
  
@app.get(f"{settings.API\_V1\_PREFIX}/financials/trial-balance")  
async def get\_trial\_balance(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session), start: str = Query(...), end: str = Query(...)):  
 org\_id, user = deps  
 data = await trial\_balance(db, org\_id=org\_id, start=datetime.fromisoformat(start), end=datetime.fromisoformat(end))  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="REPORT", resource="TrialBalance", resource\_id="\*", details={"start": start, "end": end})  
 return JSONResponse(data)  
  
@app.get(f"{settings.API\_V1\_PREFIX}/financials/pnl")  
async def get\_pnl(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session), start: str = Query(...), end: str = Query(...)):  
 org\_id, user = deps  
 data = await income\_statement(db, org\_id=org\_id, start=datetime.fromisoformat(start), end=datetime.fromisoformat(end))  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="REPORT", resource="IncomeStatement", resource\_id="\*", details={"start": start, "end": end})  
 return JSONResponse(data)  
  
@app.post(f"{settings.API\_V1\_PREFIX}/vat/returns/prepare")  
async def prepare\_vat(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session), start: str = Query(...), end: str = Query(...)):  
 org\_id, user = deps  
 result = await prepare\_vat\_return(db, org\_id=org\_id, start=datetime.fromisoformat(start), end=datetime.fromisoformat(end))  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="PREPARE", resource="VATReturn", resource\_id=str(result["id"]), details={"start": start, "end": end})  
 return JSONResponse(result)  
  
@app.get(f"{settings.API\_V1\_PREFIX}/vat/form201/preview")  
async def form201\_preview(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session), start: str = Query(...), end: str = Query(...)):  
 org\_id, user = deps  
 result = await preview\_form201(db, org\_id=org\_id, start=datetime.fromisoformat(start), end=datetime.fromisoformat(end))  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="REPORT", resource="Form201", resource\_id="\*", details={"start": start, "end": end})  
 return JSONResponse(result)

if **name** == “**main**”: uvicorn.run(“app.main:app”, host=“0.0.0.0”, port=8000, reload=True)

python  
import uvicorn  
from fastapi import FastAPI, Depends, HTTPException, Header  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select  
from .config import settings  
from .db import Base, engine, get\_session  
from .auth import get\_current\_user, require\_membership, require\_superadmin, router as auth\_router  
from .schemas import PartnerCreate, PartnerOut, InvoiceCreate, InvoiceOut, ValidationResult  
from .services.partner\_service import PartnerService  
from .services.invoice\_service import InvoiceService  
from .peppol.provider\_client import ProviderClient  
from .middleware.request\_context import RequestContextMiddleware  
from .utils.audit import log\_audit  
from .models import Organization  
from .enums import OrgStatus  
  
app = FastAPI(title=settings.APP\_NAME)  
app.add\_middleware(RequestContextMiddleware)  
app.include\_router(auth\_router)  
  
@app.on\_event("startup")  
async def startup():  
 async with engine.begin() as conn:  
 await conn.run\_sync(Base.metadata.create\_all)  
  
# Helpers  
async def org\_guard(org\_id: int = Header(..., alias="X-Org-ID"), user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_membership(org\_id, user, db)  
 org = (await db.execute(select(Organization).where(Organization.id == org\_id))).scalar\_one()  
 if org.status != OrgStatus.APPROVED:  
 raise HTTPException(status\_code=403, detail="Organization not approved yet")  
 return org\_id, user  
  
# Admin endpoints  
@app.get(f"{settings.API\_V1\_PREFIX}/admin/orgs/pending")  
async def list\_pending\_orgs(user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_superadmin(user, db)  
 rows = (await db.execute(select(Organization).where(Organization.status == OrgStatus.PENDING))).scalars().all()  
 return [{"id": o.id, "name": o.name, "status": o.status.value} for o in rows]  
  
@app.post(f"{settings.API\_V1\_PREFIX}/admin/orgs/{{org\_id}}/approve")  
async def approve\_org(org\_id: int, user=Depends(get\_current\_user), db: AsyncSession = Depends(get\_session)):  
 await require\_superadmin(user, db)  
 org = (await db.execute(select(Organization).where(Organization.id == org\_id))).scalar\_one\_or\_none()  
 if not org:  
 raise HTTPException(status\_code=404, detail="Org not found")  
 org.status = OrgStatus.APPROVED  
 await db.commit()  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="APPROVE", resource="Organization", resource\_id=org\_id, details={})  
 return {"id": org.id, "status": org.status.value}  
  
# Partners  
@app.post(f"{settings.API\_V1\_PREFIX}/partners", response\_model=PartnerOut)  
async def create\_partner(payload: PartnerCreate, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 try:  
 partner = await PartnerService.create(db, payload, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="CREATE", resource="Partner", resource\_id=partner.id, details=payload.dict())  
 return partner  
 except ValueError as e:  
 raise HTTPException(status\_code=400, detail=str(e))  
  
@app.get(f"{settings.API\_V1\_PREFIX}/partners", response\_model=list[PartnerOut])  
async def list\_partners(deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 items = await PartnerService.list\_all(db, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="LIST", resource="Partner", resource\_id="\*", details={})  
 return items  
  
# Invoices  
@app.post(f"{settings.API\_V1\_PREFIX}/invoices", response\_model=InvoiceOut)  
async def create\_invoice(payload: InvoiceCreate, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 vr = await InvoiceService.validate(db, payload, org\_id=org\_id)  
 if not vr.valid:  
 raise HTTPException(status\_code=400, detail=vr.errors)  
 inv = await InvoiceService.create(db, payload, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="CREATE", resource="Invoice", resource\_id=inv.id, details=payload.dict())  
 return inv  
  
@app.get(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}", response\_model=InvoiceOut)  
async def get\_invoice(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 inv = await InvoiceService.get(db, invoice\_id, org\_id=org\_id)  
 if not inv:  
 raise HTTPException(status\_code=404, detail="Invoice not found")  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="READ", resource="Invoice", resource\_id=invoice\_id, details={})  
 return inv  
  
@app.post(f"{settings.API\_V1\_PREFIX}/invoices/validate", response\_model=ValidationResult)  
async def validate\_invoice(payload: InvoiceCreate, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 vr = await InvoiceService.validate(db, payload, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="VALIDATE", resource="Invoice", resource\_id="\*", details={"errors": vr.errors})  
 return vr  
  
@app.post(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}/send", response\_model=InvoiceOut)  
async def send\_invoice(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 provider = ProviderClient()  
 try:  
 inv = await InvoiceService.send(db, invoice\_id, provider, org\_id=org\_id)  
 await log\_audit(db, user\_id=user.id, org\_id=org\_id, action="SEND", resource="Invoice", resource\_id=invoice\_id, details={"provider\_status": inv.provider\_status})  
 return inv  
 except ValueError as e:  
 raise HTTPException(status\_code=404, detail=str(e))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 uvicorn.run("app.main:app", host="0.0.0.0", port=8000, reload=True)

python import uvicorn from fastapi import FastAPI, Depends, HTTPException from sqlalchemy.ext.asyncio import AsyncSession from .config import settings from .db import Base, engine, get\_session from .auth import api\_key\_auth from .schemas import PartnerCreate, PartnerOut, InvoiceCreate, InvoiceOut, ValidationResult from .services.partner\_service import PartnerService from .services.invoice\_service import InvoiceService from .peppol.provider\_client import ProviderClient from .middleware.request\_context import RequestContextMiddleware

app = FastAPI(title=settings.APP\_NAME) app.add\_middleware(RequestContextMiddleware)

@app.on\_event(“startup”) async def startup(): async with engine.begin() as conn: await conn.run\_sync(Base.metadata.create\_all)

# Partners

@app.post(f”{settings.API\_V1\_PREFIX}/partners”, response\_model=PartnerOut, dependencies=[Depends(api\_key\_auth)]) async def create\_partner(payload: PartnerCreate, db: AsyncSession = Depends(get\_session)): try: return await PartnerService.create(db, payload) except ValueError as e: raise HTTPException(status\_code=400, detail=str(e))

@app.get(f”{settings.API\_V1\_PREFIX}/partners”, response\_model=list[PartnerOut], dependencies=[Depends(api\_key\_auth)]) async def list\_partners(db: AsyncSession = Depends(get\_session)): return await PartnerService.list\_all(db)

# Invoices

@app.post(f”{settings.API\_V1\_PREFIX}/invoices”, response\_model=InvoiceOut, dependencies=[Depends(api\_key\_auth)]) async def create\_invoice(payload: InvoiceCreate, db: AsyncSession = Depends(get\_session)): vr = await InvoiceService.validate(db, payload) if not vr.valid: raise HTTPException(status\_code=400, detail=vr.errors) return await InvoiceService.create(db, payload)

@app.get(f”{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}“, response\_model=InvoiceOut, dependencies=[Depends(api\_key\_auth)]) async def get\_invoice(invoice\_id: int, db: AsyncSession = Depends(get\_session)): inv = await InvoiceService.get(db, invoice\_id) if not inv: raise HTTPException(status\_code=404, detail=”Invoice not found”) return inv

@app.post(f”{settings.API\_V1\_PREFIX}/invoices/validate”, response\_model=ValidationResult, dependencies=[Depends(api\_key\_auth)]) async def validate\_invoice(payload: InvoiceCreate, db: AsyncSession = Depends(get\_session)): return await InvoiceService.validate(db, payload)

@app.post(f”{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}/send”, response\_model=InvoiceOut, dependencies=[Depends(api\_key\_auth)]) async def send\_invoice(invoice\_id: int, db: AsyncSession = Depends(get\_session)): provider = ProviderClient() try: return await InvoiceService.send(db, invoice\_id, provider) except ValueError as e: raise HTTPException(status\_code=404, detail=str(e))

if **name** == “**main**”: uvicorn.run(“app.main:app”, host=“0.0.0.0”, port=8000, reload=True)

## tests/test\_invoice\_flow.py  
```python  
import pytest  
from httpx import AsyncClient  
from app.main import app  
from app.config import settings  
  
API\_PREFIX = f"{settings.API\_V1\_PREFIX}"  
  
@pytest.mark.asyncio  
async def test\_full\_invoice\_flow\_with\_auth():  
 async with AsyncClient(app=app, base\_url="http://test") as ac:  
 # Signup to create org + owner  
 form = {"email": "owner@example.com", "password": "secret", "org\_name": "Acme"}  
 r0 = await ac.post(f"{API\_PREFIX}/auth/signup", data=form)  
 assert r0.status\_code == 200, r0.text  
 token = r0.json()["access\_token"]  
 org\_id = r0.json()["org\_id"]  
 headers = {"Authorization": f"Bearer {token}", "X-Org-ID": str(org\_id)}  
  
 # Create seller and buyer  
 seller = {  
 "name": "Seller LLC",  
 "legal\_name": "Seller LLC",  
 "trn": "123456789012345",  
 "peppol\_participant\_id": "0088:123456789",  
 "country\_code": "AE",  
 }  
 buyer = {  
 "name": "Buyer LLC",  
 "legal\_name": "Buyer LLC",  
 "trn": "987654321098765",  
 "country\_code": "AE",  
 }  
 r1 = await ac.post(f"{API\_PREFIX}/partners", json=seller, headers=headers)  
 assert r1.status\_code == 200, r1.text  
 r2 = await ac.post(f"{API\_PREFIX}/partners", json=buyer, headers=headers)  
 assert r2.status\_code == 200, r2.text  
 seller\_id = r1.json()["id"]  
 buyer\_id = r2.json()["id"]  
  
 # Create invoice  
 invoice = {  
 "number": "INV-2001",  
 "seller\_id": seller\_id,  
 "buyer\_id": buyer\_id,  
 "items": [  
 {"description": "Consulting", "quantity": 10, "unit\_price": 100.0, "vat\_rate": 0.05}  
 ],  
 "currency": "AED",  
 "due\_days": 30  
 }  
 r3 = await ac.post(f"{API\_PREFIX}/invoices", json=invoice, headers=headers)  
 assert r3.status\_code == 200, r3.text  
 inv\_id = r3.json()["id"]  
  
 # Send invoice (simulated provider by default)  
 r4 = await ac.post(f"{API\_PREFIX}/invoices/{inv\_id}/send", headers=headers)  
 assert r4.status\_code == 200, r4.text  
 body = r4.json()  
 assert body["status"] in {"SENT", "REJECTED"}  
 assert body["provider\_message\_id"] is not None

python import asyncio import json import pytest from httpx import AsyncClient from app.main import app from app.config import settings

API\_PREFIX = f”{settings.API\_V1\_PREFIX}” HEADERS = {“X-API-Key”: settings.API\_KEY}

@pytest.mark.asyncio async def test\_full\_invoice\_flow(): async with AsyncClient(app=app, base\_url=“http://test”) as ac: # Create seller and buyer seller = { “name”: “Seller LLC”, “legal\_name”: “Seller LLC”, “trn”: “123456789012345”, “peppol\_participant\_id”: “0088:123456789”, “country\_code”: “AE”, } buyer = { “name”: “Buyer LLC”, “legal\_name”: “Buyer LLC”, “trn”: “987654321098765”, “country\_code”: “AE”, } r1 = await ac.post(f”{API\_PREFIX}/partners”, json=seller, headers=HEADERS) assert r1.status\_code == 200, r1.text seller\_id = r1.json()[“id”] r2 = await ac.post(f”{API\_PREFIX}/partners”, json=buyer, headers=HEADERS) assert r2.status\_code == 200, r2.text buyer\_id = r2.json()[“id”]

# Create invoice  
 invoice = {  
 "number": "INV-1001",  
 "seller\_id": seller\_id,  
 "buyer\_id": buyer\_id,  
 "items": [  
 {"description": "Consulting", "quantity": 10, "unit\_price": 100.0, "vat\_rate": 0.05}  
 ],  
 "currency": "AED",  
 "due\_days": 30  
 }  
 r3 = await ac.post(f"{API\_PREFIX}/invoices", json=invoice, headers=HEADERS)  
 assert r3.status\_code == 200, r3.text  
 inv\_id = r3.json()["id"]  
  
 # Send invoice (simulated provider by default)  
 r4 = await ac.post(f"{API\_PREFIX}/invoices/{inv\_id}/send", headers=HEADERS)  
 assert r4.status\_code == 200, r4.text  
 body = r4.json()  
 assert body["status"] in {"SENT", "REJECTED"}  
 assert body["provider\_message\_id"] is not None

## requirements.txt  
```txt  
fastapi==0.115.0  
uvicorn[standard]==0.30.6  
pydantic==1.10.17  
SQLAlchemy==2.0.34  
aiosqlite==0.20.0  
httpx==0.27.2  
pytest==8.3.3  
pytest-asyncio==0.24.0  
lxml==5.3.0  
python-jose[cryptography]==3.3.0  
passlib[bcrypt]==1.7.4  
python-multipart==0.0.12

txt fastapi==0.115.0 uvicorn[standard]==0.30.6 pydantic==1.10.17 SQLAlchemy==2.0.34 aiosqlite==0.20.0 httpx==0.27.2 pytest==8.3.3 pytest-asyncio==0.24.0

## pyproject.toml  
```toml  
[tool.pytest.ini\_options]  
asyncio\_mode = "auto"  
addopts = "-q"  
  
[tool.black]  
line-length = 100  
  
[tool.isort]  
profile = "black"

## Dockerfile

FROM python:3.11-slim  
WORKDIR /app  
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt  
COPY . .  
ENV PYTHONUNBUFFERED=1  
EXPOSE 8000  
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

## .gitignore

\_\_pycache\_\_/  
\*.pyc  
\*.pyo  
\*.pyd  
.env  
beam.db  
.pytest\_cache/  
.idea/  
.vscode/  
.DS\_Store

## .env.example

API\_KEY=dev-key  
DB\_URL=sqlite+aiosqlite:///./beam.db  
  
# If you have a live accredited provider, set these:  
PROVIDER\_BASE\_URL=https://api.your-peppol-provider.com  
PROVIDER\_API\_KEY=replace-me  
PROVIDER\_TIMEOUT\_SECONDS=20  
  
# UBL XSD optional validation (path inside container)  
UBL\_XSD\_PATH=  
  
# JWT config  
JWT\_SECRET=change-me  
JWT\_ALGORITHM=HS256  
ACCESS\_TOKEN\_EXPIRE\_MINUTES=60

env API\_KEY=dev-key DB\_URL=sqlite+aiosqlite:///./beam.db

# If you have a live accredited provider, set these:

PROVIDER\_BASE\_URL=https://api.your-peppol-provider.com PROVIDER\_API\_KEY=replace-me PROVIDER\_TIMEOUT\_SECONDS=20

## README.md  
```md  
# New: Digital Signatures & Hash Chain + Enriched FTA Audit File  
  
### Digital signature & hash chain  
- On invoice creation we build UBL XML, compute `curr\_hash = sha256(prev\_hash + xml\_bytes)` and store `prev\_hash`.  
- If `SIGNING\_PRIVATE\_KEY\_PEM` is configured, we produce a \*\*detached signature\*\* (RSA/ECDSA) over `(prev\_hash || xml)` and store Base64 in `signature\_b64` with optional `SIGNING\_CERT\_SERIAL`.  
- Retrieve signature/chain via: `GET /api/v1/invoices/{id}/signature`.  
  
> This implements the immutability and authenticity primitives expected by FTA e‑invoicing. For production, add certificate lifecycle (rotation, CRL/OCSP checks) and signature embedding into PDFs/PEPPOL payloads.  
  
### FTA Audit File (enriched)  
- `GET /api/v1/compliance/audit-file` ZIP now includes invoice \*\*PrevHash/Hash/Signature/CertSerial\*\* columns.  
- Use this to deliver audit artifacts and prove the chain-of-trust to inspectors.  
  
### Config  
Add to `.env`:  
```env  
SIGNING\_PRIVATE\_KEY\_PEM="""-----BEGIN PRIVATE KEY-----  
...  
-----END PRIVATE KEY-----"""  
SIGNING\_CERT\_SERIAL=YOUR\_CERT\_SERIAL  
HASH\_ALG=sha256

### Notes

* We sign the canonical UBL bytes; if you also render PDFs for customers, embed the signature or include a QR with the curr\_hash for verification.
* Keep AUTO\_APPROVE\_SIGNUPS=false in production so orgs require **SUPERADMIN** approval.

md  
# Beam — UAE E‑Invoicing / e‑Billing (PEPPOL‑ready) MVP  
  
API‑first MVP for issuing, validating (UBL 2.1), and sending invoices via accredited PEPPOL provider APIs. Now with \*\*OAuth2/JWT auth\*\*, \*\*multi‑tenant orgs/roles\*\*, \*\*super‑admin org approval flow\*\*, \*\*audit logs\*\*, and \*\*basic accounting posts\*\* (AR, Revenue, VAT Payable) on invoice issuance.  
  
## Quickstart  
```bash  
cp .env.example .env  
pip install -r requirements.txt  
uvicorn app.main:app --reload

OpenAPI: http://localhost:8000/docs

## Auth & Org Approval

* POST /api/v1/auth/signup → creates Org with status APPROVED if AUTO\_APPROVE\_SIGNUPS=true (dev), else PENDING (prod). User is OWNER of that org.
* Super admin can approve: POST /api/v1/admin/orgs/{org\_id}/approve (requires SUPERADMIN membership).
* All business endpoints require Authorization: Bearer <token> and X-Org-ID: <org\_id>; if org is not APPROVED, access is denied.

## Core Endpoints

* Partners: POST /api/v1/partners, GET /api/v1/partners
* Invoices: POST /api/v1/invoices, GET /api/v1/invoices/{id}, POST /api/v1/invoices/{id}/send, POST /api/v1/invoices/validate

## Accounting (MVP)

* On invoice issuance, system posts a journal entry:
  + **Debit** AR, **Credit** Sales Revenue and VAT Payable
* Core accounts are auto‑created per org (1100 AR, 4000 Sales, 2100 VAT, 1000 Cash)
* Next: add POST /invoices/{id}/payments to record receipts (Debit Cash / Credit AR)

## PEPPOL Provider Integration

* Configure PROVIDER\_BASE\_URL + PROVIDER\_API\_KEY to enable live sends; otherwise a simulated response is returned.

## UBL 2.1 & XSD Validation

* Generate proper UBL 2.1 namespaced XML with lxml.
* If UBL\_XSD\_PATH is set to a valid UBL Invoice XSD, XML is validated before sending.

## Roles

* SUPERADMIN (platform), OWNER, ADMIN, USER (org‑scoped)

## Roadmap

* Payments endpoint + AR aging, trial balance
* Full EN 16931 checks & FTA ruleset
* Provider‑specific adapters

md  
# Beam — UAE E‑Invoicing / e‑Billing (PEPPOL‑ready) MVP  
  
API‑first MVP for issuing, validating (UBL 2.1), and sending invoices via accredited PEPPOL provider APIs. Now with \*\*OAuth2/JWT auth\*\*, \*\*multi‑tenant orgs/roles\*\*, and \*\*audit logs\*\*.  
  
## Quickstart  
```bash  
cp .env.example .env  
pip install -r requirements.txt  
uvicorn app.main:app --reload  
# or docker  
# docker build -t beam:dev . && docker run -p 8000:8000 --env-file .env beam:dev

OpenAPI: http://localhost:8000/docs

## Auth

1. **Sign up & create org**

curl -X POST http://localhost:8000/api/v1/auth/signup \  
 -F email=owner@example.com -F password=secret -F org\_name=Acme

Use returned access\_token as Authorization: Bearer <token> and X-Org-ID: <org\_id> header.

1. **Login**

curl -X POST http://localhost:8000/api/v1/auth/token -F email=owner@example.com -F password=secret

## Endpoints (core)

* POST /api/v1/partners — create partner (JWT + X-Org-ID)
* GET /api/v1/partners — list partners
* POST /api/v1/invoices — validate + create an invoice (DRAFT)
* GET /api/v1/invoices/{id} — fetch invoice
* POST /api/v1/invoices/{id}/send — build **UBL 2.1** XML + optional XSD validation + send via provider

## UBL 2.1 & XSD Validation

* We generate UBL 2.1 Invoice with namespaces using **lxml**.
* If UBL\_XSD\_PATH is set (e.g., to an official UBL XSD), the XML is validated before sending.

## Multi‑tenant & Audit

* Orgs, Users, Memberships (roles: OWNER/ADMIN/USER)
* All CRUD actions are recorded in audit\_logs

## Roadmap

* Replace optional XSD with **official UBL 2.1 schema set** and add EN 16931 checks
* Add receiving inbox & acknowledgments, payment links, analytics
* Provider-specific adapters when you pick an accredited AP

md  
# Beam — UAE E‑Invoicing / e‑Billing (PEPPOL‑ready) MVP  
  
This is an API‑first MVP for issuing, validating, and sending invoices through an accredited PEPPOL provider. It’s designed around the UAE context (VAT/TRN, AED currency) and a partnership model (you integrate via Access Point provider APIs rather than operating your own AP).  
  
## Why this Architecture  
- \*\*API‑first\*\*: easy to embed in ERP/CRM/PSP systems.  
- \*\*Provider adapter\*\*: swap any accredited provider without changing your domain logic.  
- \*\*UAE‑aware\*\*: TRN checks, AED default, expandable compliance rules.  
  
## Quickstart  
```bash  
cp .env.example .env  
# (optional) edit PROVIDER\_\* if you have a live endpoint  
  
# local  
pip install -r requirements.txt  
uvicorn app.main:app --reload  
  
# docker  
docker build -t beam:dev .  
docker run -p 8000:8000 --env-file .env beam:dev

OpenAPI docs at: http://localhost:8000/docs

## Auth

Include header: X-API-Key: <API\_KEY>

## Core Endpoints

* POST /api/v1/partners — create partner (seller/buyer)
* GET /api/v1/partners — list partners
* POST /api/v1/invoices — validate + create an invoice (DRAFT)
* GET /api/v1/invoices/{id} — fetch invoice
* POST /api/v1/invoices/{id}/send — build XML + send via provider

## PEPPOL Provider Integration

* Configure PROVIDER\_BASE\_URL and PROVIDER\_API\_KEY in .env.
* If not configured, the adapter **simulates** a success response for rapid local dev.

## Compliance Notes (UAE + PEPPOL)

* **TRN** currently validated as 15 digits (simplified). Extend with full UAE FTA rules.
* **XML builder** is a **minimal PEPPOL‑BIS style** for MVP; replace with full UBL 2.1 + XSD validation before prod.
* Add archival (WORM storage), retention periods, and data‑residency controls for production.

## Roadmap (from SWOT → Differentiation)

1. **Compliance‑as‑API**: expand /invoices/validate into full rules engine (FTA + PEPPOL), publish error codes.
2. **Network effects**: add inbox endpoints to **receive** invoices and acknowledgments, with message tracking.
3. **Payments**: attach UAE PSP payment links and auto‑reconciliation webhooks.
4. **Analytics**: out‑of‑box dashboards for error rates, delivery SLAs, DSO.
5. **Multi‑tenant**: scoped auth, orgs, roles, audit logs (SOX‑style).
6. **Security**: JWT / OAuth, KMS at‑rest encryption, key rotation, AS4/EN 16931 conformance.

## Testing

pytest -q

## License

Apache‑2.0

---  
  
# 🔧 How this aligns with your SWOT/positioning  
- \*\*Strengths\*\*: API‑first, PEPPOL adapter, UAE defaults, simple onboarding for SMEs.  
- \*\*Weaknesses addressed\*\*: adds tests, CI hooks (via `pyproject.toml`), structure for fast iteration; clear provider swap.  
- \*\*Opportunities\*\*: extend validators → compliance analytics; add payments; offer white‑label.  
- \*\*Threats mitigations\*\*: adapter indirection shields you from provider changes; config‑driven.  
  
> You can now hand this to your AI agent. It’s a complete scaffold with real endpoints, storage, simulated provider send, and tests. As you pick an accredited provider, implement the real request/response shapes inside `app/peppol/provider\_client.py` and swap the XML builder with a full UBL implementation.  
  
  
## app/utils/audit.py  
```python  
import json  
from sqlalchemy.ext.asyncio import AsyncSession  
from ..models import AuditLog  
  
async def log\_audit(db: AsyncSession, \*, user\_id: int, org\_id: int, action: str, resource: str, resource\_id: str, details: dict | None = None):  
 entry = AuditLog(  
 user\_id=user\_id,  
 org\_id=org\_id,  
 action=action,  
 resource=resource,  
 resource\_id=str(resource\_id),  
 details=json.dumps(details or {})  
 )  
 db.add(entry)  
 await db.commit()

## app/reports/audit\_export.py

import io  
import csv  
import zipfile  
from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select, and\_  
from ..models import Invoice, Partner, JournalEntry, JournalLine, Account  
  
# Build an FTA-style Audit File (CSV set) inside a zip, now including hash/chain fields  
async def build\_audit\_zip(db: AsyncSession, \*, org\_id: int, start: datetime, end: datetime) -> tuple[bytes, str]:  
 mem = io.BytesIO()  
 with zipfile.ZipFile(mem, mode="w", compression=zipfile.ZIP\_DEFLATED) as zf:  
 # Invoices.csv  
 inv\_buf = io.StringIO()  
 inv\_writer = csv.writer(inv\_buf)  
 inv\_writer.writerow([  
 "InvoiceID","Number","Type","IssueDate","SellerID","BuyerID","Currency","Subtotal","VAT","Total","Status","PrevHash","Hash","SignatureB64","CertSerial"  
 ])   
 rows = (await db.execute(select(Invoice).where(and\_(Invoice.org\_id==org\_id, Invoice.issue\_date>=start, Invoice.issue\_date<=end)))).scalars().all()  
 for i in rows:  
 inv\_writer.writerow([  
 i.id,i.number,i.invoice\_type,i.issue\_date.isoformat(),i.seller\_id,i.buyer\_id,i.currency,i.subtotal,i.vat\_amount,i.total,i.status.value,i.prev\_hash or "",i.curr\_hash or "",i.signature\_b64 or "",i.signing\_cert\_serial or ""  
 ])  
 zf.writestr("Invoices.csv", inv\_buf.getvalue())  
  
 # Partners.csv  
 p\_buf = io.StringIO()  
 p\_writer = csv.writer(p\_buf)  
 p\_writer.writerow(["PartnerID","Name","TRN","PEPPOL","Country"])  
 p\_rows = (await db.execute(select(Partner).where(Partner.org\_id==org\_id))).scalars().all()  
 for p in p\_rows:  
 p\_writer.writerow([p.id,p.name,p.trn,p.peppol\_participant\_id or "",p.country\_code])  
 zf.writestr("Partners.csv", p\_buf.getvalue())  
  
 # Journal.csv  
 j\_buf = io.StringIO()  
 j\_writer = csv.writer(j\_buf)  
 j\_writer.writerow(["EntryID","Ref","CreatedAt"])  
 j\_rows = (await db.execute(select(JournalEntry).where(and\_(JournalEntry.org\_id==org\_id, JournalEntry.created\_at>=start, JournalEntry.created\_at<=end)))).scalars().all()  
 for j in j\_rows:  
 j\_writer.writerow([j.id,j.ref,j.created\_at.isoformat()])  
 zf.writestr("Journal.csv", j\_buf.getvalue())  
  
 # JournalLines.csv  
 jl\_buf = io.StringIO()  
 jl\_writer = csv.writer(jl\_buf)  
 jl\_writer.writerow(["EntryID","AccountCode","Debit","Credit"])   
 for j in j\_rows:  
 lines = (await db.execute(select(JournalLine, Account).where(JournalLine.entry\_id==j.id, JournalLine.account\_id==Account.id))).all()  
 for jl, acc in lines:  
 jl\_writer.writerow([j.id, acc.code, jl.debit, jl.credit])  
 zf.writestr("JournalLines.csv", jl\_buf.getvalue())  
  
 filename = f"FTA\_Audit\_{org\_id}\_{start.date()}\_{end.date()}.zip"  
 return mem.getvalue(), filename

python import io import csv import zipfile from datetime import datetime from sqlalchemy.ext.asyncio import AsyncSession from sqlalchemy import select, and\_ from ..models import Invoice, Partner, JournalEntry, JournalLine, Account

# Build a simple FTA-style Audit File (CSV set) inside a zip

async def build\_audit\_zip(db: AsyncSession, \*, org\_id: int, start: datetime, end: datetime) -> tuple[bytes, str]: mem = io.BytesIO() with zipfile.ZipFile(mem, mode=“w”, compression=zipfile.ZIP\_DEFLATED) as zf: # Invoices.csv inv\_buf = io.StringIO() inv\_writer = csv.writer(inv\_buf) inv\_writer.writerow([“InvoiceID”,“Number”,“IssueDate”,“SellerID”,“BuyerID”,“Currency”,“Subtotal”,“VAT”,“Total”,“Status”]) rows = (await db.execute(select(Invoice).where(and\_(Invoice.org\_id==org\_id, Invoice.issue\_date>=start, Invoice.issue\_date<=end)))).scalars().all() for i in rows: inv\_writer.writerow([i.id,i.number,i.issue\_date.isoformat(),i.seller\_id,i.buyer\_id,i.currency,i.subtotal,i.vat\_amount,i.total,i.status.value]) zf.writestr(“Invoices.csv”, inv\_buf.getvalue())

# Partners.csv  
 p\_buf = io.StringIO()  
 p\_writer = csv.writer(p\_buf)  
 p\_writer.writerow(["PartnerID","Name","TRN","PEPPOL","Country"])  
 p\_rows = (await db.execute(select(Partner).where(Partner.org\_id==org\_id))).scalars().all()  
 for p in p\_rows:  
 p\_writer.writerow([p.id,p.name,p.trn,p.peppol\_participant\_id or "",p.country\_code])  
 zf.writestr("Partners.csv", p\_buf.getvalue())  
  
 # Journal.csv  
 j\_buf = io.StringIO()  
 j\_writer = csv.writer(j\_buf)  
 j\_writer.writerow(["EntryID","Ref","CreatedAt"])  
 j\_rows = (await db.execute(select(JournalEntry).where(and\_(JournalEntry.org\_id==org\_id, JournalEntry.created\_at>=start, JournalEntry.created\_at<=end)))).scalars().all()  
 for j in j\_rows:  
 j\_writer.writerow([j.id,j.ref,j.created\_at.isoformat()])  
 zf.writestr("Journal.csv", j\_buf.getvalue())  
  
 # JournalLines.csv  
 jl\_buf = io.StringIO()  
 jl\_writer = csv.writer(jl\_buf)  
 jl\_writer.writerow(["EntryID","AccountCode","Debit","Credit"])   
 # join lines with accounts  
 for j in j\_rows:  
 lines = (await db.execute(select(JournalLine, Account).where(JournalLine.entry\_id==j.id, JournalLine.account\_id==Account.id))).all()  
 for jl, acc in lines:  
 jl\_writer.writerow([j.id, acc.code, jl.debit, jl.credit])  
 zf.writestr("JournalLines.csv", jl\_buf.getvalue())  
  
filename = f"FTA\_Audit\_{org\_id}\_{start.date()}\_{end.date()}.zip"  
return mem.getvalue(), filename

## app/accounting/financials.py  
```python  
from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select, and\_  
from ..models import JournalEntry, JournalLine, Account, AccountType  
  
async def trial\_balance(db: AsyncSession, \*, org\_id: int, start: datetime, end: datetime) -> dict:  
 rows = (await db.execute(select(JournalEntry).where(and\_(JournalEntry.org\_id==org\_id, JournalEntry.created\_at>=start, JournalEntry.created\_at<=end)))).scalars().all()  
 totals: dict[str, dict] = {}  
 for j in rows:  
 lines = (await db.execute(select(JournalLine, Account).where(JournalLine.entry\_id==j.id, JournalLine.account\_id==Account.id))).all()  
 for jl, acc in lines:  
 if acc.code not in totals:  
 totals[acc.code] = {"account": acc.name, "type": acc.type, "debit": 0.0, "credit": 0.0}  
 totals[acc.code]["debit"] += jl.debit  
 totals[acc.code]["credit"] += jl.credit  
 return {"org\_id": org\_id, "start": start.isoformat(), "end": end.isoformat(), "totals": totals}  
  
async def income\_statement(db: AsyncSession, \*, org\_id: int, start: datetime, end: datetime) -> dict:  
 tb = await trial\_balance(db, org\_id=org\_id, start=start, end=end)  
 revenue = 0.0  
 expense = 0.0  
 for code, row in tb["totals"].items():  
 if row["type"] == AccountType.REVENUE:  
 revenue += row["credit"] - row["debit"]  
 if row["type"] == AccountType.EXPENSE:  
 expense += row["debit"] - row["credit"]  
 net = revenue - expense  
 return {"org\_id": org\_id, "start": start.isoformat(), "end": end.isoformat(), "revenue": revenue, "expense": expense, "net\_income": net}

## app/vat/returns.py

from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select, and\_  
from ..models import Invoice, VATReturn, VATReturnStatus  
  
# MVP: only Output VAT from issued invoices; Input VAT (purchases) = 0 placeholder  
async def prepare\_vat\_return(db: AsyncSession, \*, org\_id: int, start: datetime, end: datetime) -> dict:  
 rows = (await db.execute(select(Invoice).where(and\_(Invoice.org\_id==org\_id, Invoice.issue\_date>=start, Invoice.issue\_date<=end)))).scalars().all()  
 output\_vat = sum(i.vat\_amount for i in rows)  
 input\_vat = 0.0  
 net = output\_vat - input\_vat  
  
 vr = VATReturn(org\_id=org\_id, period\_start=start, period\_end=end, output\_vat=output\_vat, input\_vat=input\_vat, net\_vat=net, status=VATReturnStatus.DRAFT)  
 db.add(vr)  
 await db.commit()  
 await db.refresh(vr)  
 return {"id": vr.id, "org\_id": org\_id, "period\_start": start.isoformat(), "period\_end": end.isoformat(), "output\_vat": output\_vat, "input\_vat": input\_vat, "net\_vat": net, "status": vr.status.value}

## app/archiving/retention.py

# Simple retention policy placeholder (extend to enforce WORM storage, S3/GCS backends, etc.)  
from datetime import timedelta  
  
RETENTION\_YEARS = 7 # FTA recommends 5+; set 7 to be safe  
  
def get\_retention\_policy():  
 return {"years": RETENTION\_YEARS, "notes": "Keep invoice & accounting records for at least the retention period; ensure immutability."}

## tests/test\_compliance.py

import pytest  
from httpx import AsyncClient  
from app.main import app  
from app.config import settings  
  
API\_PREFIX = f"{settings.API\_V1\_PREFIX}"  
  
@pytest.mark.asyncio  
async def test\_audit\_file\_and\_vat\_return():  
 async with AsyncClient(app=app, base\_url="http://test") as ac:  
 # Signup  
 r0 = await ac.post(f"{API\_PREFIX}/auth/signup", data={"email": "cfo@example.com", "password": "secret", "org\_name": "BeamCo"})  
 assert r0.status\_code == 200  
 token = r0.json()["access\_token"]  
 org\_id = r0.json()["org\_id"]  
 headers = {"Authorization": f"Bearer {token}", "X-Org-ID": str(org\_id)}  
  
 # Seed minimal partners & invoice  
 seller = {"name": "Seller LLC","legal\_name": "Seller LLC","trn": "123456789012345","peppol\_participant\_id": "0088:123456789","country\_code": "AE"}  
 buyer = {"name": "Buyer LLC","legal\_name": "Buyer LLC","trn": "987654321098765","country\_code": "AE"}  
 S = await ac.post(f"{API\_PREFIX}/partners", json=seller, headers=headers); assert S.status\_code == 200  
 B = await ac.post(f"{API\_PREFIX}/partners", json=buyer, headers=headers); assert B.status\_code == 200  
 inv = {"number": "INV-3001","seller\_id": S.json()["id"],"buyer\_id": B.json()["id"],"items": [{"description": "Service","quantity": 1,"unit\_price": 100.0,"vat\_rate": 0.05}],"currency": "AED","due\_days": 30}  
 C = await ac.post(f"{API\_PREFIX}/invoices", json=inv, headers=headers); assert C.status\_code == 200  
  
 # Audit file export (zip)  
 A = await ac.get(f"{API\_PREFIX}/compliance/audit-file?start=2025-01-01T00:00:00&end=2025-12-31T23:59:59", headers=headers)  
 assert A.status\_code == 200  
 assert A.headers.get("Content-Type").startswith("application/zip")  
  
 # VAT return prepare (JSON)  
 V = await ac.post(f"{API\_PREFIX}/vat/returns/prepare?start=2025-01-01T00:00:00&end=2025-12-31T23:59:59", headers=headers)  
 assert V.status\_code == 200  
 data = V.json()  
 assert data["output\_vat"] >= 5.0

## README.md

# Compliance Modules Added  
  
- \*\*Audit File Export (FTA‑style)\*\* — `GET /api/v1/compliance/audit-file?start=ISO8601&end=ISO8601` returns a \*\*ZIP\*\* containing CSVs: Invoices, Partners, Journal, JournalLines. Use this to satisfy FTA audit data requests.  
- \*\*Financials\*\* — Trial Balance and P&L JSON APIs:  
 - `GET /api/v1/financials/trial-balance?start&end`  
 - `GET /api/v1/financials/pnl?start&end`  
- \*\*VAT Returns (MVP)\*\* — `POST /api/v1/vat/returns/prepare?start&end` computes \*\*Output VAT\*\* from issued invoices (Input VAT = 0 placeholder). Stores a `vat\_returns` record with status `DRAFT`.  
- \*\*Archiving/Retention\*\* — Config scaffold (`RETENTION\_YEARS = 7`), ready to wire to immutable storage.  
  
> Note: This is a \*compliance‑oriented MVP\*. For full FTA compliance, add: official FAF spec export, input‑VAT (purchases) ingestion, Form‑201 mapping, WORM storage & data residency controls, and auditor access roles. The scaffolding here is intentionally designed so each of those can be slotted in cleanly.

## app/peppol/validators.py

from decimal import Decimal, ROUND\_HALF\_UP  
from typing import List, Optional  
from datetime import datetime  
from pydantic import BaseModel  
from sqlalchemy import select  
from sqlalchemy.ext.asyncio import AsyncSession  
  
from ..models import Invoice  
from ..schemas import ValidationErrorItem, ValidationErrorCode, ValidationResult, InvoiceCreate  
  
VAT\_ROUND\_PLACES = Decimal("0.01")  
  
  
def quant\_round(value: Decimal) -> Decimal:  
 return value.quantize(VAT\_ROUND\_PLACES, rounding=ROUND\_HALF\_UP)  
  
  
def \_is\_valid\_trn(trn: Optional[str]) -> bool:  
 if not trn:  
 return False  
 # Simple TRN validation: digits only, typical UAE TRN lengths between 9 and 15  
 return trn.isdigit() and 9 <= len(trn) <= 15  
  
  
async def validate\_invoice\_business\_rules(payload: InvoiceCreate, db: Optional[AsyncSession] = None, org\_id: Optional[int] = None) -> ValidationResult:  
 errors: List[ValidationErrorItem] = []  
  
 # Basic presence checks  
 if not payload.items or len(payload.items) == 0:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.MISSING\_MANDATORY\_UBL, message="Invoice must contain at least one line item", field="items"))  
  
 # Line-level checks  
 subtotal = Decimal("0.00")  
 vat\_total = Decimal("0.00")  
 for idx, item in enumerate(payload.items):  
 try:  
 qty = Decimal(str(item.quantity))  
 price = Decimal(str(item.unit\_price))  
 rate = Decimal(str(item.vat\_rate))  
 except Exception:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.MISSING\_MANDATORY\_UBL, message=f"Invalid numeric on line {idx}", field=f"items[{idx}]") )  
 continue  
 if qty <= 0:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.NEGATIVE\_LINE, message=f"Quantity must be > 0 on line {idx}", field=f"items[{idx}].quantity"))  
 if price < 0:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.NEGATIVE\_LINE, message=f"Unit price must be >= 0 on line {idx}", field=f"items[{idx}].unit\_price"))  
 line\_net = qty \* price  
 line\_vat = (line\_net \* rate)  
 subtotal += line\_net  
 vat\_total += line\_vat  
  
 # Rounding checks  
 calc\_subtotal = quant\_round(subtotal)  
 calc\_vat = quant\_round(vat\_total)  
 calc\_total = quant\_round(calc\_subtotal + calc\_vat)  
  
 # If invoice carries declared totals (some clients may pre-fill), compare  
 # Pydantic payload may not include subtotal/vat\_amount fields; use optional attributes if present  
 declared\_sub = getattr(payload, "declared\_subtotal", None)  
 declared\_vat = getattr(payload, "declared\_vat", None)  
 declared\_total = getattr(payload, "declared\_total", None)  
 if declared\_sub is not None or declared\_vat is not None or declared\_total is not None:  
 try:  
 if declared\_sub is not None and quant\_round(Decimal(str(declared\_sub))) != calc\_subtotal:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.ROUNDING\_ERROR, message="Declared subtotal mismatches calculated subtotal", field="declared\_subtotal"))  
 if declared\_vat is not None and quant\_round(Decimal(str(declared\_vat))) != calc\_vat:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.ROUNDING\_ERROR, message="Declared VAT mismatches calculated VAT", field="declared\_vat"))  
 if declared\_total is not None and quant\_round(Decimal(str(declared\_total))) != calc\_total:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.ROUNDING\_ERROR, message="Declared total mismatches calculated total", field="declared\_total"))  
 except Exception:  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.MISSING\_MANDATORY\_UBL, message="Invalid declared totals", field="\*"))  
  
 # TRN checks - ensure both seller and buyer TRNs exist in partners table if db provided  
 if db and org\_id:  
 # Seller and buyer existence will be verified elsewhere, but we check invoice number uniqueness here  
 res = await db.execute(select(Invoice).where(Invoice.number == payload.number, Invoice.org\_id == org\_id))  
 if res.scalars().first():  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.DUPLICATE\_NUMBER, message="Invoice number already exists for org", field="number"))  
  
 # Date checks  
 if payload.issue\_date:  
 if payload.issue\_date > datetime.utcnow():  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.INVALID\_DATE, message="Issue date cannot be in the future", field="issue\_date"))  
  
 # Mandatory UBL items placeholder (e.g., InvoiceTypeCode)  
 if not getattr(payload, "invoice\_type", None):  
 errors.append(ValidationErrorItem(code=ValidationErrorCode.MISSING\_MANDATORY\_UBL, message="Missing invoice\_type (SALE/PURCHASE/CREDIT\_NOTE)", field="invoice\_type"))  
  
 # Return aggregated result  
 return ValidationResult(valid=(len(errors) == 0), errors=errors)

## app/peppol/xml\_builder.py (enhanced UBL generation + strict validation)

from lxml import etree  
from datetime import datetime, timedelta  
from decimal import Decimal  
from ..schemas import InvoiceCreate  
from ..config import settings  
  
# Compact but fuller UBL 2.1 generator — include InvoiceTypeCode, TaxTotal, TaxSubtotal, UBLExtensions  
NSMAP = {  
 None: "urn:oasis:names:specification:ubl:schema:xsd:Invoice-2",  
 "cbc": "urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2",  
 "cac": "urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2",  
}  
  
  
def \_money\_str(value: Decimal) -> str:  
 return f"{value.quantize(Decimal('0.01')):.2f}"  
  
  
def build\_invoice\_xml(payload: InvoiceCreate, seller\_trn: str, buyer\_trn: str) -> bytes:  
 issue\_date = payload.issue\_date or datetime.utcnow()  
 due\_date = issue\_date + timedelta(days=payload.due\_days)  
  
 inv = etree.Element("Invoice", nsmap=NSMAP)  
  
 # UBL extensions (placeholder for signing metadata)  
 ube = etree.SubElement(inv, "UBLExtensions")  
 etree.SubElement(ube, "UBLExtension")  
  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}UBLVersionID").text = "2.1"  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}CustomizationID").text = "urn:fdc:peppol.eu:poacc:billing:3"  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}ProfileID").text = "urn:fdc:peppol.eu:poacc:billing:3:ver3.0"  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}ID").text = payload.number  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}IssueDate").text = issue\_date.strftime("%Y-%m-%d")  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}DueDate").text = due\_date.strftime("%Y-%m-%d")  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}InvoiceTypeCode").text = getattr(payload, "invoice\_type", "SALE")  
 etree.SubElement(inv, f"{{{NSMAP['cbc']}}}DocumentCurrencyCode").text = payload.currency.value  
  
 # Supplier Party  
 asp = etree.SubElement(inv, f"{{{NSMAP['cac']}}}AccountingSupplierParty")  
 party = etree.SubElement(asp, f"{{{NSMAP['cac']}}}Party")  
 pid = etree.SubElement(party, f"{{{NSMAP['cac']}}}PartyIdentification")  
 etree.SubElement(pid, f"{{{NSMAP['cbc']}}}ID").text = seller\_trn  
  
 # Customer Party  
 acp = etree.SubElement(inv, f"{{{NSMAP['cac']}}}AccountingCustomerParty")  
 party2 = etree.SubElement(acp, f"{{{NSMAP['cac']}}}Party")  
 pid2 = etree.SubElement(party2, f"{{{NSMAP['cac']}}}PartyIdentification")  
 etree.SubElement(pid2, f"{{{NSMAP['cbc']}}}ID").text = buyer\_trn  
  
 # Invoice lines  
 subtotal = Decimal("0.00")  
 vat\_total = Decimal("0.00")  
 for idx, item in enumerate(payload.items, start=1):  
 il = etree.SubElement(inv, f"{{{NSMAP['cac']}}}InvoiceLine")  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}ID").text = str(idx)  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}InvoicedQuantity").text = str(item.quantity)  
 line\_amount = Decimal(str(item.quantity)) \* Decimal(str(item.unit\_price))  
 etree.SubElement(il, f"{{{NSMAP['cbc']}}}LineExtensionAmount", currencyID=payload.currency.value).text = \_money\_str(line\_amount)  
  
 item\_node = etree.SubElement(il, f"{{{NSMAP['cac']}}}Item")  
 etree.SubElement(item\_node, f"{{{NSMAP['cbc']}}}Description").text = item.description  
  
 price = etree.SubElement(il, f"{{{NSMAP['cac']}}}Price")  
 etree.SubElement(price, f"{{{NSMAP['cbc']}}}PriceAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(str(item.unit\_price)))  
  
 subtotal += line\_amount  
 vat\_total += line\_amount \* Decimal(str(item.vat\_rate))  
  
 # Tax totals  
 tax\_total = etree.SubElement(inv, f"{{{NSMAP['cac']}}}TaxTotal")  
 etree.SubElement(tax\_total, f"{{{NSMAP['cbc']}}}TaxAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(vat\_total))  
  
 # Tax subtotal (single rate assumption for MVP)  
 tax\_sub = etree.SubElement(tax\_total, f"{{{NSMAP['cac']}}}TaxSubtotal")  
 etree.SubElement(tax\_sub, f"{{{NSMAP['cbc']}}}TaxableAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(subtotal))  
 etree.SubElement(tax\_sub, f"{{{NSMAP['cbc']}}}TaxAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(vat\_total))  
 tax\_category = etree.SubElement(tax\_sub, f"{{{NSMAP['cac']}}}TaxCategory")  
 etree.SubElement(tax\_category, f"{{{NSMAP['cbc']}}}ID").text = "S" # Standard  
 tax\_scheme = etree.SubElement(tax\_category, f"{{{NSMAP['cac']}}}TaxScheme")  
 etree.SubElement(tax\_scheme, f"{{{NSMAP['cbc']}}}ID").text = "VAT"  
  
 # Monetary totals  
 lmt = etree.SubElement(inv, f"{{{NSMAP['cac']}}}LegalMonetaryTotal")  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}LineExtensionAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(subtotal))  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}TaxExclusiveAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(subtotal))  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}TaxAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(vat\_total))  
 etree.SubElement(lmt, f"{{{NSMAP['cbc']}}}PayableAmount", currencyID=payload.currency.value).text = \_money\_str(Decimal(subtotal + vat\_total))  
  
 xml\_bytes = etree.tostring(inv, xml\_declaration=True, encoding="UTF-8")  
  
 # Strict XSD validation if path provided  
 if settings.UBL\_XSD\_PATH:  
 try:  
 with open(settings.UBL\_XSD\_PATH, "rb") as f:  
 schema\_doc = etree.parse(f)  
 schema = etree.XMLSchema(schema\_doc)  
 schema.assertValid(etree.fromstring(xml\_bytes))  
 except Exception as e:  
 # Convert to ValueError so callers can respond with 400  
 raise ValueError(f"UBL XSD validation failed: {e}")  
  
 return xml\_bytes

## app/utils/crypto.py (add verification)

import base64  
import hashlib  
from typing import Optional  
from cryptography.hazmat.primitives import hashes, serialization  
from cryptography.hazmat.primitives.asymmetric import padding, rsa, ec  
from cryptography.hazmat.primitives.asymmetric.utils import Prehashed  
from cryptography.hazmat.backends import default\_backend  
  
ALG\_MAP = {  
 "sha256": hashes.SHA256,  
}  
  
def sha256(data: bytes) -> str:  
 return hashlib.sha256(data).hexdigest()  
  
def load\_private\_key(pem: str):  
 return serialization.load\_pem\_private\_key(pem.encode(), password=None, backend=default\_backend())  
  
def load\_public\_key\_from\_pem(pem: str):  
 return serialization.load\_pem\_public\_key(pem.encode(), backend=default\_backend())  
  
def sign\_detached(data: bytes, private\_key\_pem: Optional[str], hash\_alg: str = "sha256") -> Optional[str]:  
 if not private\_key\_pem:  
 return None  
 key = load\_private\_key(private\_key\_pem)  
 hclass = ALG\_MAP.get(hash\_alg, hashes.SHA256)  
 if isinstance(key, rsa.RSAPrivateKey):  
 sig = key.sign(data, padding.PKCS1v15(), hclass())  
 elif isinstance(key, ec.EllipticCurvePrivateKey):  
 sig = key.sign(data, ec.ECDSA(hclass()))  
 else:  
 raise ValueError("Unsupported key type")  
 return base64.b64encode(sig).decode()  
  
def verify\_signature(data: bytes, signature\_b64: str, public\_key\_pem: str, hash\_alg: str = "sha256") -> bool:  
 sig = base64.b64decode(signature\_b64)  
 key = load\_public\_key\_from\_pem(public\_key\_pem)  
 hclass = ALG\_MAP.get(hash\_alg, hashes.SHA256)  
 try:  
 if isinstance(key, rsa.RSAPublicKey):  
 key.verify(sig, data, padding.PKCS1v15(), hclass())  
 elif isinstance(key, ec.EllipticCurvePublicKey):  
 key.verify(sig, data, ec.ECDSA(hclass()))  
 else:  
 return False  
 return True  
 except Exception:  
 return False

## app/services/invoice\_service.py (wrap create in atomic DB transaction and use stricter validator)

import json  
from datetime import datetime  
from sqlalchemy.ext.asyncio import AsyncSession  
from sqlalchemy import select, and\_  
from ..models import Invoice, Partner, Account, AccountType, JournalEntry, JournalLine  
from ..schemas import InvoiceCreate, ValidationResult  
from ..enums import InvoiceStatus  
from ..peppol.validators import validate\_invoice\_business\_rules  
from ..peppol.xml\_builder import build\_invoice\_xml  
from ..peppol.provider\_client import ProviderClient  
from ..config import settings  
from ..utils.crypto import sha256, sign\_detached  
  
async def ensure\_core\_accounts(db: AsyncSession, org\_id: int) -> dict[str, Account]:  
 async def get\_or\_create(code: str, name: str, typ: str) -> Account:  
 res = await db.execute(select(Account).where(Account.org\_id == org\_id, Account.code == code))  
 acc = res.scalar\_one\_or\_none()  
 if not acc:  
 acc = Account(org\_id=org\_id, code=code, name=name, type=typ)  
 db.add(acc)  
 await db.flush()  
 return acc  
 ar = await get\_or\_create("1100", "Accounts Receivable", AccountType.ASSET)  
 ap = await get\_or\_create("2100", "Accounts Payable", AccountType.LIABILITY)  
 rev = await get\_or\_create("4000", "Sales Revenue", AccountType.REVENUE)  
 vat\_out = await get\_or\_create("2200", "VAT Payable (Output)", AccountType.LIABILITY)  
 vat\_in = await get\_or\_create("1300", "VAT Receivable (Input)", AccountType.ASSET)  
 cash = await get\_or\_create("1000", "Cash", AccountType.ASSET)  
 exp = await get\_or\_create("5000", "Expenses", AccountType.EXPENSE)  
 await db.commit()  
 return {"AR": ar, "AP": ap, "REV": rev, "VAT\_OUT": vat\_out, "VAT\_IN": vat\_in, "CASH": cash, "EXP": exp}  
  
class InvoiceService:  
 @staticmethod  
 async def create(db: AsyncSession, payload: InvoiceCreate, \*, org\_id: int) -> Invoice:  
 # Validate business rules (may require DB for duplicates)  
 vr = await validate\_invoice\_business\_rules(payload, db=db, org\_id=org\_id)  
 if not vr.valid:  
 raise ValueError({"validation\_errors": [e.dict() for e in vr.errors]})  
  
 subtotal = sum([i.quantity \* i.unit\_price for i in payload.items])  
 vat\_amount = sum([i.quantity \* i.unit\_price \* i.vat\_rate for i in payload.items])  
 sign\_factor = -1.0 if payload.invoice\_type in {"CREDIT\_NOTE\_SALE", "CREDIT\_NOTE\_PURCHASE"} else 1.0  
 subtotal \*= sign\_factor  
 vat\_amount \*= sign\_factor  
 total = subtotal + vat\_amount  
  
 invoice = Invoice(  
 org\_id=org\_id,  
 number=payload.number,  
 invoice\_type=payload.invoice\_type,  
 seller\_id=payload.seller\_id,  
 buyer\_id=payload.buyer\_id,  
 issue\_date=payload.issue\_date or datetime.utcnow(),  
 due\_days=payload.due\_days,  
 currency=payload.currency.value,  
 subtotal=subtotal,  
 vat\_amount=vat\_amount,  
 total=total,  
 status=InvoiceStatus.DRAFT,  
 items\_json=json.dumps([i.dict() for i in payload.items]),  
 meta\_json=json.dumps({"notes": payload.notes or ""}),  
 )  
  
 # Wrap creation, hashing/signing, and journal posting in a transaction  
 async with db.begin():  
 db.add(invoice)  
 await db.flush()  
  
 # Hash chain  
 prev = (await db.execute(select(Invoice).where(and\_(Invoice.org\_id==org\_id, Invoice.curr\_hash!=None)).order\_by(Invoice.issue\_date.desc(), Invoice.id.desc()))).scalars().first()  
 prev\_hash = prev.curr\_hash if prev else None  
  
 seller = await db.get(Partner, invoice.seller\_id)  
 buyer = await db.get(Partner, invoice.buyer\_id)  
 payload\_for\_xml = InvoiceCreate(  
 number=invoice.number,  
 seller\_id=invoice.seller\_id,  
 buyer\_id=invoice.buyer\_id,  
 invoice\_type=invoice.invoice\_type,  
 issue\_date=invoice.issue\_date,  
 due\_days=invoice.due\_days,  
 items=json.loads(invoice.items\_json),  
 currency=invoice.currency,  
 )  
 xml\_bytes = build\_invoice\_xml(payload\_for\_xml, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 to\_hash = (prev\_hash or "").encode() + xml\_bytes  
 curr\_hash = sha256(to\_hash)  
 signature\_b64 = sign\_detached(to\_hash, settings.SIGNING\_PRIVATE\_KEY\_PEM, settings.HASH\_ALG)  
  
 invoice.prev\_hash = prev\_hash  
 invoice.curr\_hash = curr\_hash  
 invoice.signature\_b64 = signature\_b64  
 invoice.signing\_cert\_serial = settings.SIGNING\_CERT\_SERIAL  
  
 # Journal posting  
 accts = await ensure\_core\_accounts(db, org\_id)  
 je = JournalEntry(org\_id=org\_id, ref=invoice.number, memo=f"Invoice {payload.invoice\_type}")  
 db.add(je)  
 await db.flush()  
  
 if payload.invoice\_type in {"SALE", "CREDIT\_NOTE\_SALE"}:  
 db.add\_all([  
 JournalLine(entry\_id=je.id, account\_id=accts["AR"].id, debit=total if total>0 else 0.0, credit=abs(total) if total<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["REV"].id, debit=abs(subtotal) if subtotal<0 else 0.0, credit=subtotal if subtotal>0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["VAT\_OUT"].id, debit=abs(vat\_amount) if vat\_amount<0 else 0.0, credit=vat\_amount if vat\_amount>0 else 0.0),  
 ])  
 else:  
 expense\_amount = subtotal  
 db.add\_all([  
 JournalLine(entry\_id=je.id, account\_id=accts["EXP"].id, debit=expense\_amount if expense\_amount>0 else 0.0, credit=abs(expense\_amount) if expense\_amount<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["VAT\_IN"].id, debit=vat\_amount if vat\_amount>0 else 0.0, credit=abs(vat\_amount) if vat\_amount<0 else 0.0),  
 JournalLine(entry\_id=je.id, account\_id=accts["AP"].id, debit=abs(total) if total<0 else 0.0, credit=total if total>0 else 0.0),  
 ])  
  
 # End transaction (commit or rollback automatically)  
 await db.refresh(invoice)  
 return invoice

## app/main.py (signature verification endpoint)

from fastapi import HTTPException  
from ..utils.crypto import verify\_signature, load\_public\_key\_from\_pem  
from ..config import settings  
  
@app.get(f"{settings.API\_V1\_PREFIX}/invoices/{{invoice\_id}}/verify")  
async def verify\_invoice(invoice\_id: int, deps=Depends(org\_guard), db: AsyncSession = Depends(get\_session)):  
 org\_id, user = deps  
 inv = await InvoiceService.get(db, invoice\_id, org\_id=org\_id)  
 if not inv:  
 raise HTTPException(status\_code=404, detail="Invoice not found")  
 if not inv.curr\_hash:  
 return {"ok": False, "reason": "Invoice not hashed/signed"}  
 if not inv.signature\_b64:  
 return {"ok": False, "reason": "No signature available"}  
  
 # Recreate canonical xml and to\_hash using prev\_hash  
 seller = await db.get(Partner, inv.seller\_id)  
 buyer = await db.get(Partner, inv.buyer\_id)  
 payload\_for\_xml = InvoiceCreate(  
 number=inv.number,  
 seller\_id=inv.seller\_id,  
 buyer\_id=inv.buyer\_id,  
 invoice\_type=inv.invoice\_type,  
 issue\_date=inv.issue\_date,  
 due\_days=inv.due\_days,  
 items=json.loads(inv.items\_json),  
 currency=inv.currency,  
 )  
 xml\_bytes = build\_invoice\_xml(payload\_for\_xml, seller\_trn=seller.trn, buyer\_trn=buyer.trn)  
 to\_hash = (inv.prev\_hash or "").encode() + xml\_bytes  
 recomputed = sha256(to\_hash)  
 if recomputed != inv.curr\_hash:  
 return {"ok": False, "reason": "Hash mismatch", "expected": inv.curr\_hash, "computed": recomputed}  
  
 # Verify signature — requires public key; for MVP we expect public key stored in ENV as SIGNING\_PUBLIC\_KEY\_PEM  
 pub\_pem = getattr(settings, "SIGNING\_PUBLIC\_KEY\_PEM", None)  
 if not pub\_pem:  
 return {"ok": False, "reason": "No public key configured for verification"}  
 verified = verify\_signature(to\_hash, inv.signature\_b64, pub\_pem, settings.HASH\_ALG)  
 return {"ok": bool(verified), "reason": None if verified else "Signature verification failed"}

## tests/test\_validation\_and\_signing.py

import pytest  
from httpx import AsyncClient  
from app.main import app  
from app.config import settings  
  
API\_PREFIX = f"{settings.API\_V1\_PREFIX}"  
  
@pytest.mark.asyncio  
async def test\_validation\_rejects\_duplicate\_and\_invalid\_lines():  
 async with AsyncClient(app=app, base\_url="http://test") as ac:  
 r0 = await ac.post(f"{API\_PREFIX}/auth/signup", data={"email": "val@example.com", "password": "secret", "org\_name": "ValCo"})  
 token = r0.json()["access\_token"]  
 org\_id = r0.json()["org\_id"]  
 headers = {"Authorization": f"Bearer {token}", "X-Org-ID": str(org\_id)}  
  
 seller = {"name": "S","trn": "12345678901","country\_code": "AE"}  
 buyer = {"name": "B","trn": "98765432101","country\_code": "AE"}  
 S = await ac.post(f"{API\_PREFIX}/partners", json=seller, headers=headers); assert S.status\_code == 200  
 B = await ac.post(f"{API\_PREFIX}/partners", json=buyer, headers=headers); assert B.status\_code == 200  
  
 inv = {"number": "INV-DUP-1","seller\_id": S.json()["id"],"buyer\_id": B.json()["id"],"items": [],"currency": "AED","due\_days": 30}  
 r = await ac.post(f"{API\_PREFIX}/invoices", json=inv, headers=headers)  
 assert r.status\_code == 400  
  
@pytest.mark.asyncio  
async def test\_signature\_and\_verify\_workflow():  
 # This test assumes SIGNING\_PRIVATE\_KEY\_PEM and SIGNING\_PUBLIC\_KEY\_PEM are set in env for test runner  
 async with AsyncClient(app=app, base\_url="http://test") as ac:  
 r0 = await ac.post(f"{API\_PREFIX}/auth/signup", data={"email": "sig@example.com", "password": "secret", "org\_name": "SigCo"})  
 token = r0.json()["access\_token"]  
 org\_id = r0.json()["org\_id"]  
 headers = {"Authorization": f"Bearer {token}", "X-Org-ID": str(org\_id)}  
  
 seller = {"name": "S","trn": "12345678901","country\_code": "AE"}  
 buyer = {"name": "B","trn": "98765432101","country\_code": "AE"}  
 S = await ac.post(f"{API\_PREFIX}/partners", json=seller, headers=headers); assert S.status\_code == 200  
 B = await ac.post(f"{API\_PREFIX}/partners", json=buyer, headers=headers); assert B.status\_code == 200  
  
 inv = {"number": "INV-SIGN-1","seller\_id": S.json()["id"],"buyer\_id": B.json()["id"],"items": [{"description": "Svc","quantity": 1,"unit\_price": 100.0,"vat\_rate": 0.05}],"currency": "AED","due\_days": 30}  
 C = await ac.post(f"{API\_PREFIX}/invoices", json=inv, headers=headers); assert C.status\_code == 200  
 iid = C.json()["id"]  
 V = await ac.get(f"{API\_PREFIX}/invoices/{iid}/verify", headers=headers)  
 assert V.status\_code == 200  
 body = V.json()  
 # If public key missing, ok: expect a structured message; if present, ok to be verified True  
 assert "ok" in body