Purple Squirrel Cookbook

Solutions & Patterns for Every Problem

Part I: Claude/AI Recipes

Recipe 1.1: Structured Output Extraction

Problem: You need Claude to return data in a specific format that your code can parse.

Solution: Use explicit format instructions with examples.

```
import anthropic

client = anthropic.Anthropic()

def extract_entities(text: str) -> dict:
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=1024,
        messages=[{
            "role": "user",
            "content": f"""Extract entities from this text and return as JSON.

Text: {text}

Return format:
{{
    "people": ["name1", "name2"],
    "companies": ["company1"],
    "dates": ["2024-01-15"],
    "amounts": ["$1,000"]
}}

Return ONLY valid JSON, no explanation."""
        }]
    )

    import json
    return json.loads(response.content[0].text)
```

Variation: For complex schemas, provide a complete example in the prompt.

Recipe 1.2: Multi-turn Conversation with Memory

Problem: Maintain context across multiple exchanges.

Solution: Pass the full conversation history.

```python
class Conversation:
    def __init__(self):
        self.client = anthropic.Anthropic()
        self.messages = []
        self.system = "You are a helpful coding assistant."

    def chat(self, user_input: str) -> str:
        self.messages.append({
            "role": "user",
            "content": user_input
        })

        response = self.client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=4096,
            system=self.system,
            messages=self.messages
        )

        assistant_message = response.content[0].text
        self.messages.append({
            "role": "assistant",
            "content": assistant_message
        })

        return assistant_message

    def clear(self):
        self.messages = []
```

Usage:

```python
conv = Conversation()
conv.chat("How do I read a file in Python?")
conv.chat("Now show me how to write to it")  # Remembers context
```

Recipe 1.3: Tool Use / Function Calling

Problem: Let Claude execute functions based on user requests.

Solution: Define tools and handle tool calls.

```python
import anthropic
import json

tools = [
    {
        "name": "get_weather",
        "description": "Get current weather for a location",
        "input_schema": {
```

```
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City name"
                }
            },
            "required": ["location"]
        }
    }
]

def get_weather(location: str) -> dict:
    # Actual API call would go here
    return {"temp": 72, "condition": "sunny"}

def process_with_tools(user_input: str):
    client = anthropic.Anthropic()

    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=1024,
        tools=tools,
        messages=[{"role": "user", "content": user_input}]
    )

    # Check if Claude wants to use a tool
    for block in response.content:
        if block.type == "tool_use":
            if block.name == "get_weather":
                result = get_weather(block.input["location"])
                # Send result back to Claude
                return continue_with_tool_result(
                    response, block.id, result
                )

    return response.content[0].text
```

Recipe 1.4: RAG Implementation

Problem: Answer questions using your own documents.

Solution: Retrieve relevant chunks, include in context.

```
from sentence_transformers import SentenceTransformer
import numpy as np

class SimpleRAG:
    def __init__(self):
        self.encoder = SentenceTransformer('all-MiniLM-L6-v2')
        self.documents = []
```

```
        self.embeddings = None

    def add_documents(self, docs: list[str]):
        self.documents.extend(docs)
        self.embeddings = self.encoder.encode(self.documents)

    def retrieve(self, query: str, top_k: int = 3) -> list[str]:
        query_embedding = self.encoder.encode([query])[0]

        # Cosine similarity
        similarities = np.dot(self.embeddings, query_embedding)
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        return [self.documents[i] for i in top_indices]

    def answer(self, question: str) -> str:
        relevant_docs = self.retrieve(question)
        context = "\n\n".join(relevant_docs)

        client = anthropic.Anthropic()
        response = client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=1024,
            messages=[{
                "role": "user",
                "content": f"""Answer based on this context:

{context}

Question: {question}

If the answer isn't in the context, say "I don't have that information."""
            }]
        )
        return response.content[0].text
```

Recipe 1.5: Automated Code Review

Problem: Get AI feedback on code changes.

Solution: Pass diff to Claude with review instructions.

```
def review_code(diff: str, context: str = "") -> dict:
    client = anthropic.Anthropic()

    prompt = f"""Review this code diff and provide feedback.

{f"Context: {context}" if context else ""}

```diff
{diff}
```
```

Analyze for:

1. Bugs or logic errors
2. Security issues
3. Performance concerns
4. Code style/readability
5. Missing edge cases

Return as JSON: {{ "summary": "One sentence overview", "issues": [{{ "severity": "high|medium|low", "line": 42, "issue": "Description", "suggestion": "How to fix" }}], "approved": true/false }}"""

```
response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=2048,
    messages=[{"role": "user", "content": prompt}]
)

import json
return json.loads(response.content[0].text)
```

````
---

# Part II: Full-Stack Recipes

## Recipe 2.1: JWT Authentication System

**Problem:** Implement secure user authentication.

**Solution:** JWT with refresh tokens.

```python
# auth.py
from datetime import datetime, timedelta
from typing import Optional
import jwt
from passlib.context import CryptContext

SECRET_KEY = "your-secret-key"  # Use env variable
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE = timedelta(minutes=15)
REFRESH_TOKEN_EXPIRE = timedelta(days=7)

pwd_context = CryptContext(schemes=["bcrypt"])

def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)

def create_access_token(user_id: str) -> str:
    expire = datetime.utcnow() + ACCESS_TOKEN_EXPIRE
````
````

```
    payload = {
        "sub": user_id,
        "exp": expire,
        "type": "access"
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def create_refresh_token(user_id: str) -> str:
    expire = datetime.utcnow() + REFRESH_TOKEN_EXPIRE
    payload = {
        "sub": user_id,
        "exp": expire,
        "type": "refresh"
    }
    return jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(token: str) -> Optional[dict]:
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None
```

FastAPI Integration:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer

security = HTTPBearer()

async def get_current_user(credentials = Depends(security)):
    token = credentials.credentials
    payload = verify_token(token)

    if not payload or payload.get("type") != "access":
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token"
        )

    return payload["sub"]
```

Recipe 2.2: WebSocket Real-time Updates

Problem: Push live updates to connected clients.

Solution: WebSocket with connection management.

```python
# websocket_manager.py
from fastapi import WebSocket
from typing import Dict, Set
import json

class ConnectionManager:
    def __init__(self):
        # room_id -> set of websockets
        self.rooms: Dict[str, Set[WebSocket]] = {}

    async def connect(self, websocket: WebSocket, room_id: str):
        await websocket.accept()
        if room_id not in self.rooms:
            self.rooms[room_id] = set()
        self.rooms[room_id].add(websocket)

    def disconnect(self, websocket: WebSocket, room_id: str):
        if room_id in self.rooms:
            self.rooms[room_id].discard(websocket)

    async def broadcast(self, room_id: str, message: dict):
        if room_id not in self.rooms:
            return
        data = json.dumps(message)
        for websocket in self.rooms[room_id]:
            try:
                await websocket.send_text(data)
            except:
                self.rooms[room_id].discard(websocket)

manager = ConnectionManager()
```

Endpoint:

```python
from fastapi import FastAPI, WebSocket, WebSocketDisconnect

app = FastAPI()

@app.websocket("/ws/{room_id}")
async def websocket_endpoint(websocket: WebSocket, room_id: str):
    await manager.connect(websocket, room_id)
    try:
        while True:
            data = await websocket.receive_json()
            # Handle incoming message
            await manager.broadcast(room_id, {
                "type": "message",
                "data": data
            })
```

```
    except WebSocketDisconnect:
        manager.disconnect(websocket, room_id)
```

Client (JavaScript):

```
const ws = new WebSocket('ws://localhost:8000/ws/room123');

ws.onmessage = (event) => {
  const data = JSON.parse(event.data);
  console.log('Received:', data);
};

ws.send(JSON.stringify({ message: 'Hello!' }));
```

Recipe 2.3: File Upload with Processing

Problem: Handle file uploads with validation and processing.

Solution: Chunked upload with background processing.

```
from fastapi import FastAPI, UploadFile, BackgroundTasks
from pathlib import Path
import hashlib
import aiofiles

UPLOAD_DIR = Path("uploads")
MAX_SIZE = 10 * 1024 * 1024  # 10MB
ALLOWED_TYPES = {"image/jpeg", "image/png", "application/pdf"}

async def save_upload(file: UploadFile) -> str:
    # Validate content type
    if file.content_type not in ALLOWED_TYPES:
        raise ValueError(f"Invalid file type: {file.content_type}")

    # Generate unique filename
    content = await file.read()
    if len(content) > MAX_SIZE:
        raise ValueError("File too large")

    file_hash = hashlib.sha256(content).hexdigest()[:16]
    ext = Path(file.filename).suffix
    filename = f"{file_hash}{ext}"
    filepath = UPLOAD_DIR / filename

    # Save file
    async with aiofiles.open(filepath, 'wb') as f:
        await f.write(content)

    return filename

def process_file(filename: str):
```

```
    """Background task to process uploaded file"""
    filepath = UPLOAD_DIR / filename
    # Resize image, extract text from PDF, etc.
    print(f"Processing {filename}")

@app.post("/upload")
async def upload_file(
    file: UploadFile,
    background_tasks: BackgroundTasks
):
    try:
        filename = await save_upload(file)
        background_tasks.add_task(process_file, filename)
        return {"filename": filename, "status": "processing"}
    except ValueError as e:
        raise HTTPException(400, str(e))
```

Recipe 2.4: Full-Text Search

Problem: Implement search across text content.

Solution: PostgreSQL full-text search (simple) or Elasticsearch (advanced).

PostgreSQL Approach:

```
-- Add search vector column
ALTER TABLE articles ADD COLUMN search_vector tsvector;

-- Create index
CREATE INDEX articles_search_idx ON articles USING GIN(search_vector);

-- Update trigger
CREATE OR REPLACE FUNCTION articles_search_trigger() RETURNS trigger AS $$
BEGIN
  NEW.search_vector :=
    setweight(to_tsvector('english', COALESCE(NEW.title, '')), 'A') ||
    setweight(to_tsvector('english', COALESCE(NEW.content, '')), 'B');
  RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER articles_search_update
  BEFORE INSERT OR UPDATE ON articles
  FOR EACH ROW EXECUTE FUNCTION articles_search_trigger();
```

Python Query:

```
from sqlalchemy import text

def search_articles(query: str, limit: int = 20):
    # Sanitize and format query
```

```
    search_query = " & ".join(query.split())

    result = db.execute(text("""
        SELECT id, title, ts_rank(search_vector, query) as rank
        FROM articles, plainto_tsquery('english', :query) query
        WHERE search_vector @@ query
        ORDER BY rank DESC
        LIMIT :limit
    """), {"query": query, "limit": limit})

    return result.fetchall()
```

Recipe 2.5: Stripe Payment Integration

Problem: Accept payments with Stripe.

Solution: Checkout Sessions for one-time, Subscriptions for recurring.

```
import stripe
from fastapi import FastAPI, Request

stripe.api_key = "sk_test_..."

@app.post("/create-checkout-session")
async def create_checkout(price_id: str, user_id: str):
    session = stripe.checkout.Session.create(
        payment_method_types=["card"],
        line_items=[{
            "price": price_id,
            "quantity": 1,
        }],
        mode="subscription",  # or "payment" for one-time
        success_url="https://yoursite.com/success?session_id={CHECKOUT_SESSION_ID}",
        cancel_url="https://yoursite.com/canceled",
        client_reference_id=user_id,
    )
    return {"url": session.url}

@app.post("/webhook")
async def stripe_webhook(request: Request):
    payload = await request.body()
    sig = request.headers.get("stripe-signature")

    try:
        event = stripe.Webhook.construct_event(
            payload, sig, "whsec_..."
        )
    except Exception as e:
        raise HTTPException(400, str(e))

    if event["type"] == "checkout.session.completed":
```

```
        session = event["data"]["object"]
        user_id = session["client_reference_id"]
        # Activate subscription for user

    elif event["type"] == "customer.subscription.deleted":
        # Handle cancellation
        pass

    return {"status": "ok"}
```

Part III: DevOps/Infrastructure

Recipe 3.1: Docker Multi-Stage Build

Problem: Create minimal production Docker images.

Solution: Multi-stage builds to separate build and runtime.

```
# Dockerfile

# Stage 1: Build
FROM node:20-alpine AS builder
WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# Stage 2: Production
FROM node:20-alpine AS production
WORKDIR /app

# Non-root user
RUN addgroup -g 1001 -S app && \
    adduser -S -D -H -u 1001 -h /app -s /sbin/nologin -G app app

COPY --from=builder --chown=app:app /app/dist ./dist
COPY --from=builder --chown=app:app /app/node_modules ./node_modules
COPY --from=builder --chown=app:app /app/package.json ./

USER app
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

Python Variant:

```
FROM python:3.11-slim AS builder
WORKDIR /app
```

```
RUN pip install poetry
COPY pyproject.toml poetry.lock ./
RUN poetry export -f requirements.txt > requirements.txt
RUN pip wheel --no-cache-dir --wheel-dir /wheels -r requirements.txt

FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /wheels /wheels
RUN pip install --no-cache-dir /wheels/*
COPY . .
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0"]
```

Recipe 3.2: GitHub Actions CI/CD

Problem: Automate testing and deployment.

Solution: Workflow with test, build, and deploy stages.

```
# .github/workflows/deploy.yml
name: Deploy

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

      - name: Run linter
        run: npm run lint
```

```yaml
  build:
    needs: test
    runs-on: ubuntu-latest
    if: github.event_name == 'push'
    permissions:
      contents: read
      packages: write

    steps:
      - uses: actions/checkout@v4

      - name: Log in to Container registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          push: true
          tags: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}

  deploy:
    needs: build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - name: Deploy to production
        run: |
          curl -X POST ${{ secrets.DEPLOY_WEBHOOK }} \
            -H "Authorization: Bearer ${{ secrets.DEPLOY_TOKEN }}" \
            -d '{"image": "${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{ github.sha }}"}'
```

Recipe 3.3: Railway/Vercel Deployment

Problem: Deploy without managing infrastructure.

Solution: Platform-specific configurations.

Vercel (Next.js):

```
// vercel.json
{
  "framework": "nextjs",
  "regions": ["iad1"],
  "env": {
    "DATABASE_URL": "@database-url"
  },
```

```
  "headers": [
    {
      "source": "/api/(.*)",
      "headers": [
        { "key": "Cache-Control", "value": "no-store" }
      ]
    }
  ]
}
```

Railway (Docker):

```
# railway.toml
[build]
builder = "dockerfile"

[deploy]
healthcheckPath = "/health"
healthcheckTimeout = 30
restartPolicyType = "on-failure"
restartPolicyMaxRetries = 3
```

Railway with Nixpacks:

```
// nixpacks.json (alternative to Dockerfile)
{
  "providers": ["python"],
  "phases": {
    "setup": {
      "nixPkgs": ["python311", "gcc"]
    },
    "install": {
      "cmds": ["pip install -r requirements.txt"]
    }
  },
  "start": "uvicorn main:app --host 0.0.0.0 --port $PORT"
}
```

Recipe 3.4: Prometheus + Grafana Monitoring

Problem: Monitor application health and performance.

Solution: Expose metrics, collect with Prometheus, visualize in Grafana.

Python Metrics Endpoint:

```
from prometheus_client import Counter, Histogram, generate_latest
from fastapi import FastAPI, Response

REQUEST_COUNT = Counter(
    'http_requests_total',
```

```
    'Total HTTP requests',
    ['method', 'endpoint', 'status']
)

REQUEST_LATENCY = Histogram(
    'http_request_duration_seconds',
    'HTTP request latency',
    ['method', 'endpoint']
)

@app.middleware("http")
async def metrics_middleware(request, call_next):
    start = time.time()
    response = await call_next(request)
    duration = time.time() - start

    REQUEST_COUNT.labels(
        method=request.method,
        endpoint=request.url.path,
        status=response.status_code
    ).inc()

    REQUEST_LATENCY.labels(
        method=request.method,
        endpoint=request.url.path
    ).observe(duration)

    return response

@app.get("/metrics")
async def metrics():
    return Response(
        generate_latest(),
        media_type="text/plain"
    )
```

Docker Compose Stack:

```
# docker-compose.monitoring.yml
services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
```

```
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

Prometheus Config:

```
# prometheus.yml
scrape_configs:
  - job_name: 'app'
    static_configs:
      - targets: ['app:8000']
    metrics_path: '/metrics'
```

Recipe 3.5: Database Migrations

Problem: Manage database schema changes safely.

Solution: Version-controlled migrations with Alembic (Python) or Prisma (TypeScript).

Alembic Setup:

```
pip install alembic
alembic init migrations
```

Migration Script:

```
# migrations/versions/001_create_users.py
"""create users table

Revision ID: 001
Create Date: 2024-01-15
"""
from alembic import op
import sqlalchemy as sa

revision = '001'
down_revision = None

def upgrade():
    op.create_table(
        'users',
        sa.Column('id', sa.UUID(), primary_key=True),
        sa.Column('email', sa.String(255), unique=True, nullable=False),
        sa.Column('name', sa.String(255)),
        sa.Column('created_at', sa.DateTime(), server_default=sa.func.now()),
    )
    op.create_index('ix_users_email', 'users', ['email'])

def downgrade():
    op.drop_index('ix_users_email')
    op.drop_table('users')
```

Run Migrations:

```
# Create new migration
alembic revision -m "add posts table"

# Run all pending migrations
alembic upgrade head

# Rollback one migration
alembic downgrade -1

# Show current version
alembic current
```

Prisma Variant (TypeScript):

```
// schema.prisma
model User {
  id        String   @id @default(uuid())
  email     String   @unique
  name      String?
  posts     Post[]
  createdAt DateTime @default(now())
}

model Post {
  id        String   @id @default(uuid())
  title     String
  content   String
  author    User     @relation(fields: [authorId], references: [id])
  authorId  String
}
```

```
# Generate migration
npx prisma migrate dev --name add_posts

# Apply to production
npx prisma migrate deploy
```
