API versioning and documentation

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

datacamp

Why API versioning?

- API endpoints as menu items
- Keep old customers happy
- Some customers want new options
- **Iterate without impacting existing customers**

API for cloud AI jobs

```python
from pydantic import BaseModel

class AIJobV1(BaseModel):
    job_name: str
    data: bytes
```

```python
class AIJobV1(BaseModel):
    job_name: str
    data: bytes
    config: dict
```

FastAPI

Versioned endpoints

```python
from pydantic import BaseModel

class AIJobV1(BaseModel):
    job_name: str
    data: bytes

class AIJobV2(BaseModel):
    job_name: str
    data: bytes
    config: dict
```

```python
from fastapi import FastAPI

app = FastAPI()

@app.post("/v1/ai-job")
def ai_job_v1(job: AIJobV1):
    ...

@app.post("/v2/ai-job")
def ai_job_v2(job: AIJobV2):
    ...
```

Reasons to update endpoint version

- Breaking change in schema
- Change in underlying function
  - Updated model code
  - Updated model training set
  - Updated pre/post processing

Iteration with optional fields

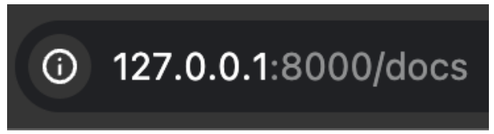
- Versioning is not always required to iterate
- Optional fields can support additional data without breaking schemas

```python
from pydantic import BaseModel

class AIJobV1(BaseModel):
    job_name: str
    data: bytes

from typing import Optional

class AIJobV1(BaseModel):
    job_name: str
    data: bytes
    config: Optional[dict]
```

Documenting APIs with Swagger

- Standard tool for API documentation
  - Keeps track of endpoints and versions
  - Built on OpenAPI standard metadata

Swagger UI



FastAPI 0.1.0 OAS 3.1

/openapi.json

AI Job API

default

POST /v1/ai-job Ai Job V1

POST /v2/ai-job Ai Job V2

Schemas

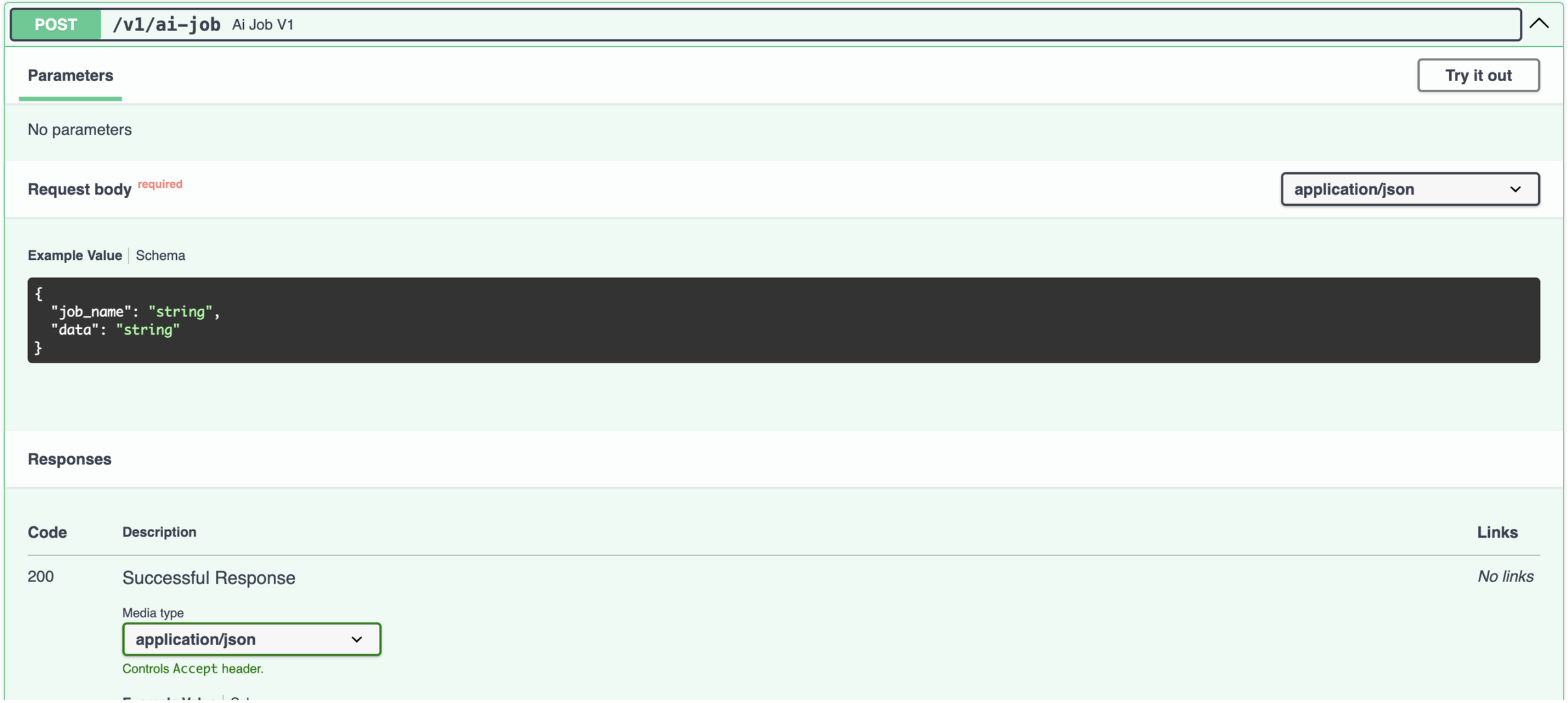
AIJobV1 > Expand all object

AIJobV2 > Expand all object

HTTPValidationError > Expand all object

ValidationError > Expand all object

Swagger UI for an endpoint



Using FastAPI's description field

```
from fastapi import FastAPI

app = FastAPI(
    description="AI Job API"
)
```

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Advanced input validation and error handling

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

Why we need advanced input

- API for restaurant orders
- Variable number of items

```
class Order(BaseModel):
    item1: str
    item2: str
    item3: str
```

Nested Pydantic models

```python
from pydantic import BaseModel

class Foo(BaseModel):
    count: int

class Bar(BaseModel):
    foo: Foo
```

```
>>> m = Bar(foo={'count': 4})
>>> print(m)
foo=Foo(count=4)
```

```python
from pydantic import BaseModel
from typing import List

class OrderItem(BaseModel):
    name: str
    quantity: int

class RestaurantOrder(BaseModel):
    customer_name: str
    items: List[OrderItem]
```

Custom model validators

```python
from fastapi import FastAPI
from fastapi.exceptions import (
    RequestValidationError
)
from pydantic import (
    BaseModel,
    model_validator,
)
from typing import List

class OrderItem(BaseModel):
    name: str
    quantity: int
```

```python
class RestaurantOrder(BaseModel):
    customer_name: str
    items: List[OrderItem]

    @model_validator(mode="after")
    def validate_after(self):
        if len(self.items) == 0:
            raise RequestValidationError(
                "No items in order!"
            )
        return self
```

```
{"detail":"No items in order!"}
```

Global exception handlers

```python
from fastapi import FastAPI
from fastapi.exceptions import RequestValidationError
from fastapi.responses import PlainTextResponse

app = FastAPI()

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    msg = "Input validation error. See the documentation: http://127.0.0.1:8000/docs"
    return PlainTextResponse(msg, status_code=422)
```

```
Input validation error. See the documentation: http://127.0.0.1:8000/docs
```

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Monitoring and logging

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

datacamp

Why monitoring and logging?

- Can't debug in production
- App supervisor needs a simple health check
- Logging key metrics over time

Setting up custom logging

- Load the uvicorn error logger
- Add custom logs to app startup
- Add custom logs to endpoints

```python
from fastapi import FastAPI
import logging

logger = logging.getLogger(
    'uvicorn.error'
)
app = FastAPI()
logger.info("App is running!")

@app.get('/')
async def main():
    logger.debug('GET /')
    return 'ok'
```

Logging a when a model is loaded

```
from fastapi import FastAPI
import logging
import joblib

logger = logging.getLogger('uvicorn.error')

model = joblib.load('penguin_classifier.pkl')
logger.info("Penguin classifier loaded successfully.")

app = FastAPI()
```

Logging process time with middleware

```
from fastapi import FastAPI, Request
import logging
import time
logger = logging.getLogger('uvicorn.error')
app = FastAPI()

@app.middleware("http")
async def log_process_time(request: Request, call_next):
    start_time = time.perf_counter()
    response = await call_next(request)
    process_time = time.perf_counter() - start_time
    logger.info(f"Process time was {process_time} seconds.")
    return response
```

[1] https://fastapi.tiangolo.com/tutorial/middleware/

Setting the logging level

Log Level	Numeric Value
debug	10
info	20
warning	30
error	40
critical	50

```
uvicorn main:app --log-level debug
```

Monitoring

```python
from fastapi import FastAPI

app = FastAPI()
@app.get("/health")
async def get_health():
    return {"status": "OK"}
```

- "I'm ok!"

Sharing model parameters with monitoring

```
from fastapi import FastAPI
import joblib

model = joblib.load(
    'penguin_classifier.pkl'
)
app = FastAPI()
@app.get("/health")
async def get_health():
    params = model.get_params()
    return {"status": "OK",
            "params": params}
```

- "I'm ok!"
- "Here are some fun facts about me!"

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Wrap-up

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

Introduction to FastAPI for Model Deployment

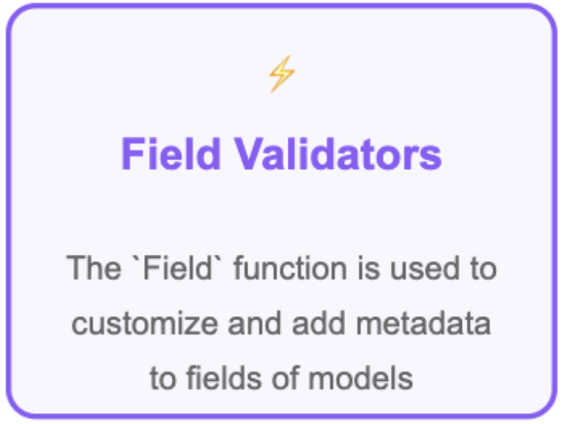
Chapter 1

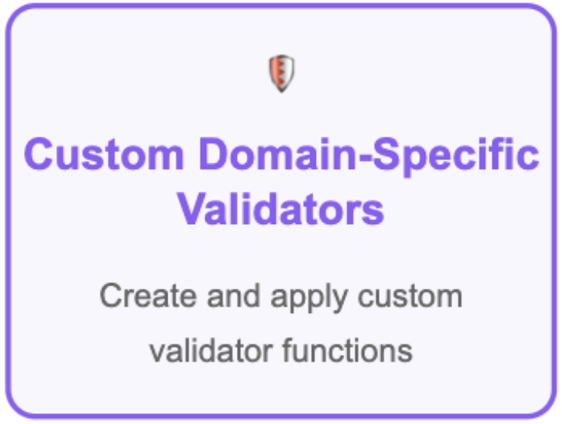
- Basic GET and POST requests
- Loading a pre-trained model
- Running the `uvicorn` server
- Pydantic models for requests and responses

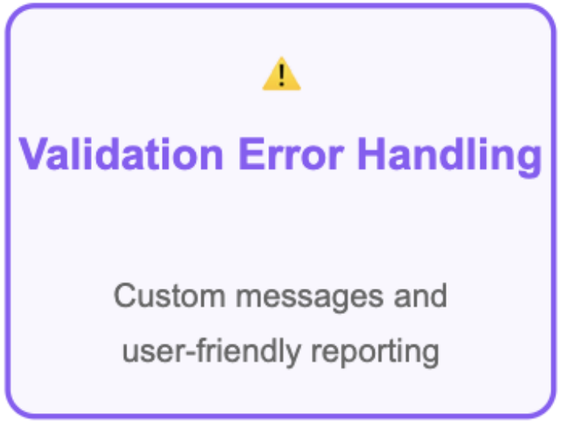
Integrating AI models

Chapter 2

- More structured input types
- Loading a pre-trained model in the app
- Structured prediction results



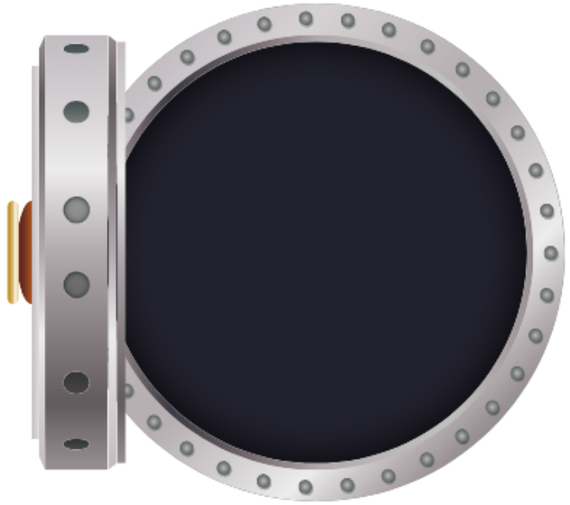




Securing and optimizing the API

Chapter 3

- API key authentication
- Rate limiting
- Async processing



API versioning, monitoring and logging

Chapter 4

- API versioning and documentation
- Advanced input validation and error handling
- Monitoring and logging

Congratulations!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI