# PytoWeb Framework Source Code

This document contains the source code of the PytoWeb Framework.

\_\_init\_\_.py

"""  
PytoWeb - Create web interfaces using pure Python  
  
This module provides a comprehensive framework for building web interfaces using pure Python.  
Version: 0.1.0  
Author: PytoWeb Team  
License: MIT  
"""  
  
from .app import App  
from .router import Router  
from .server import Server  
from .components import Component, Button, Container, Text, Link, Image  
from .layouts import Grid, Flex  
from .themes import Theme  
from .animations import (  
 Animation,  
 AnimationManager,  
 FADE\_IN,  
 FADE\_OUT,  
 SLIDE\_IN,  
 SLIDE\_OUT,  
 SLIDE\_UP,  
 SLIDE\_DOWN,  
 ROTATE,  
 SCALE,  
 BOUNCE,  
 SHAKE,  
 PULSE,  
 ELASTIC\_IN,  
 ELASTIC\_OUT,  
 SWING,  
 WOBBLE,  
 ZOOM\_IN,  
 ZOOM\_OUT  
)  
from .events import EventBridge, EventDelegate  
from .vdom import VDOMRenderer  
  
\_\_version\_\_ = "0.1.0"  
\_\_author\_\_ = "PytoWeb Team"  
\_\_license\_\_ = "MIT"  
  
\_\_all\_\_ = [  
 # Core  
 'App',  
 'Router',  
 'Server',  
 'VDOMRenderer',  
   
 # Components  
 'Component',  
 'Button',  
 'Container',  
 'Text',  
 'Link',  
 'Image',  
   
 # Layouts  
 'Grid',  
 'Flex',  
   
 # Themes  
 'Theme',  
   
 # Animations  
 'Animation',  
 'AnimationManager',  
 'FADE\_IN',  
 'FADE\_OUT',  
 'SLIDE\_IN',  
 'SLIDE\_OUT',  
 'SLIDE\_UP',  
 'SLIDE\_DOWN',  
 'ROTATE',  
 'SCALE',  
 'BOUNCE',  
 'SHAKE',  
 'PULSE',  
 'ELASTIC\_IN',  
 'ELASTIC\_OUT',  
 'SWING',  
 'WOBBLE',  
 'ZOOM\_IN',  
 'ZOOM\_OUT',  
   
 # Events  
 'EventBridge',  
 'EventDelegate'  
]

animations.py

"""  
Animation system for PytoWeb  
"""  
  
from typing import Dict, Any, List, Optional, Union, Tuple  
from dataclasses import dataclass  
from .styles import Style  
  
@dataclass  
class AnimationTiming:  
 """Animation timing configuration"""  
 duration: float = 0.3  
 delay: float = 0  
 iteration\_count: Union[int, str] = 1  
 direction: str = 'normal'  
 timing\_function: str = 'ease'  
 fill\_mode: str = 'forwards'  
  
class Animation:  
 """Animation definition class"""  
   
 def \_\_init\_\_(self, name: str, keyframes: Dict[str, Dict[str, str]], timing: Optional[AnimationTiming] = None):  
 self.name = name  
 self.keyframes = keyframes  
 self.timing = timing or AnimationTiming()  
   
 def to\_css(self) -> str:  
 """Convert animation to CSS"""  
 css = [f'@keyframes {self.name} {{']  
   
 for selector, styles in self.keyframes.items():  
 css.append(f' {selector} {{')  
 for prop, value in styles.items():  
 css.append(f' {prop}: {value};')  
 css.append(' }')  
   
 css.append('}')  
 return '\n'.join(css)  
   
 def get\_animation\_css(self) -> str:  
 """Get animation CSS properties"""  
 return (  
 f'animation: {self.name} '  
 f'{self.timing.duration}s '  
 f'{self.timing.timing\_function} '  
 f'{self.timing.delay}s '  
 f'{self.timing.iteration\_count} '  
 f'{self.timing.direction} '  
 f'{self.timing.fill\_mode}'  
 )  
  
class AnimationSequence:  
 """Animation sequence for chaining multiple animations"""  
 def \_\_init\_\_(self, \*animations: Tuple[Animation, float]):  
 self.animations = animations  
 self.total\_duration = sum(duration for \_, duration in animations)  
   
 def to\_css(self) -> str:  
 """Convert animation sequence to CSS"""  
 css\_parts = []  
 current\_time = 0  
   
 for animation, duration in self.animations:  
 start\_percent = (current\_time / self.total\_duration) \* 100  
 end\_percent = ((current\_time + duration) / self.total\_duration) \* 100  
   
 for selector, styles in animation.keyframes.items():  
 if selector == 'from':  
 selector = f'{start\_percent}%'  
 elif selector == 'to':  
 selector = f'{end\_percent}%'  
 else:  
 # 调整中间关键帧的时间点  
 original\_percent = float(selector.replace('%', ''))  
 adjusted\_percent = start\_percent + (original\_percent / 100) \* (end\_percent - start\_percent)  
 selector = f'{adjusted\_percent}%'  
   
 css\_parts.append(f' {selector} {{')  
 for prop, value in styles.items():  
 css\_parts.append(f' {prop}: {value};')  
 css\_parts.append(' }')  
   
 current\_time += duration  
   
 return '@keyframes ' + self.name + ' {\n' + '\n'.join(css\_parts) + '\n}'  
  
class Flip(Animation):  
 """3D flip animation"""  
 def \_\_init\_\_(self, direction: str = 'x', duration: float = 0.6):  
 timing = AnimationTiming(duration=duration)  
 axis = 'X' if direction.lower() == 'x' else 'Y'  
   
 super().\_\_init\_\_(f'flip-{direction}', {  
 'from': {  
 'transform': f'perspective(400px) rotate{axis}(0)',  
 'animation-timing-function': 'ease-out'  
 },  
 '40%': {  
 'transform': f'perspective(400px) translate{axis}(0) rotate{axis}(-20deg)',  
 'animation-timing-function': 'ease-out'  
 },  
 '60%': {  
 'transform': f'perspective(400px) rotate{axis}(10deg)',  
 'animation-timing-function': 'ease-in'  
 },  
 '80%': {  
 'transform': f'perspective(400px) rotate{axis}(-5deg)',  
 'animation-timing-function': 'ease-in'  
 },  
 'to': {  
 'transform': f'perspective(400px)',  
 'animation-timing-function': 'ease-in'  
 }  
 }, timing)  
  
class Elastic(Animation):  
 """Elastic animation"""  
 def \_\_init\_\_(self, direction: str = 'in', duration: float = 1.0):  
 timing = AnimationTiming(duration=duration)  
   
 if direction == 'in':  
 keyframes = {  
 '0%': {'transform': 'scale(0)'},  
 '55%': {'transform': 'scale(1.05)'},  
 '75%': {'transform': 'scale(0.95)'},  
 '90%': {'transform': 'scale(1.02)'},  
 '100%': {'transform': 'scale(1)'}  
 }  
 else: # out  
 keyframes = {  
 '0%': {'transform': 'scale(1)'},  
 '25%': {'transform': 'scale(0.95)'},  
 '50%': {'transform': 'scale(1.05)'},  
 '75%': {'transform': 'scale(0.95)'},  
 '100%': {'transform': 'scale(0)'}  
 }  
   
 super().\_\_init\_\_(f'elastic-{direction}', keyframes, timing)  
  
class Swing(Animation):  
 """Swing animation"""  
 def \_\_init\_\_(self, duration: float = 1.0):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('swing', {  
 '0%': {'transform': 'rotate(0deg)'},  
 '20%': {'transform': 'rotate(15deg)'},  
 '40%': {'transform': 'rotate(-10deg)'},  
 '60%': {'transform': 'rotate(5deg)'},  
 '80%': {'transform': 'rotate(-5deg)'},  
 '100%': {'transform': 'rotate(0deg)'}  
 }, timing)  
  
class Wobble(Animation):  
 """Wobble animation"""  
 def \_\_init\_\_(self, duration: float = 1.0):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('wobble', {  
 '0%': {'transform': 'translateX(0%)'},  
 '15%': {'transform': 'translateX(-25%) rotate(-5deg)'},  
 '30%': {'transform': 'translateX(20%) rotate(3deg)'},  
 '45%': {'transform': 'translateX(-15%) rotate(-3deg)'},  
 '60%': {'transform': 'translateX(10%) rotate(2deg)'},  
 '75%': {'transform': 'translateX(-5%) rotate(-1deg)'},  
 '100%': {'transform': 'translateX(0%)'}  
 }, timing)  
  
class TypeWriter(Animation):  
 """Typewriter text animation"""  
 def \_\_init\_\_(self, text\_length: int, duration: float = 2.0):  
 timing = AnimationTiming(duration=duration)  
 steps = {}  
   
 for i in range(text\_length + 1):  
 percentage = (i / text\_length) \* 100  
 steps[f'{percentage}%'] = {  
 'width': f'{i}ch',  
 'border-right-color': f'{"transparent" if i == text\_length else "currentColor"}'  
 }  
   
 super().\_\_init\_\_('typewriter', steps, timing)  
  
class FadeIn(Animation):  
 """Fade in animation"""  
 def \_\_init\_\_(self, duration: float = 0.3):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('fade-in', {  
 'from': {'opacity': '0'},  
 'to': {'opacity': '1'}  
 }, timing)  
  
class FadeOut(Animation):  
 """Fade out animation"""  
 def \_\_init\_\_(self, duration: float = 0.3):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('fade-out', {  
 'from': {'opacity': '1'},  
 'to': {'opacity': '0'}  
 }, timing)  
  
class Slide(Animation):  
 """Slide animation"""  
 def \_\_init\_\_(self, direction: str = 'left', duration: float = 0.3):  
 timing = AnimationTiming(duration=duration)  
   
 if direction == 'left':  
 keyframes = {  
 'from': {'transform': 'translateX(-100%)', 'opacity': '0'},  
 'to': {'transform': 'translateX(0)', 'opacity': '1'}  
 }  
 elif direction == 'right':  
 keyframes = {  
 'from': {'transform': 'translateX(100%)', 'opacity': '0'},  
 'to': {'transform': 'translateX(0)', 'opacity': '1'}  
 }  
 elif direction == 'up':  
 keyframes = {  
 'from': {'transform': 'translateY(-100%)', 'opacity': '0'},  
 'to': {'transform': 'translateY(0)', 'opacity': '1'}  
 }  
 else: # down  
 keyframes = {  
 'from': {'transform': 'translateY(100%)', 'opacity': '0'},  
 'to': {'transform': 'translateY(0)', 'opacity': '1'}  
 }  
   
 super().\_\_init\_\_(f'slide-{direction}', keyframes, timing)  
  
class Rotate(Animation):  
 """Rotate animation"""  
 def \_\_init\_\_(self, degrees: int = 360, duration: float = 0.3):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('rotate', {  
 'from': {'transform': 'rotate(0deg)'},  
 'to': {'transform': f'rotate({degrees}deg)'}  
 }, timing)  
  
class Scale(Animation):  
 """Scale animation"""  
 def \_\_init\_\_(self, from\_scale: float = 0, to\_scale: float = 1, duration: float = 0.3):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('scale', {  
 'from': {'transform': f'scale({from\_scale})'},  
 'to': {'transform': f'scale({to\_scale})'}  
 }, timing)  
  
class Bounce(Animation):  
 """Bounce animation"""  
 def \_\_init\_\_(self, duration: float = 1.0):  
 timing = AnimationTiming(duration=duration)  
 super().\_\_init\_\_('bounce', {  
 '0%': {'transform': 'translateY(0)'},  
 '20%': {'transform': 'translateY(0)'},  
 '40%': {'transform': 'translateY(-30px)'},  
 '50%': {'transform': 'translateY(0)'},  
 '60%': {'transform': 'translateY(-15px)'},  
 '80%': {'transform': 'translateY(0)'},  
 '100%': {'transform': 'translateY(0)'}  
 }, timing)  
  
class Shake(Animation):  
 """Shake animation"""  
 def \_\_init\_\_(self, intensity: int = 10, duration: float = 0.8):  
 timing = AnimationTiming(duration=duration)  
 keyframes = {}  
 steps = 10  
 for i in range(steps + 1):  
 percentage = f"{(i \* 100) // steps}%"  
 if i % 2 == 0:  
 keyframes[percentage] = {'transform': f'translateX({intensity}px)'}  
 else:  
 keyframes[percentage] = {'transform': f'translateX(-{intensity}px)'}  
 keyframes['100%'] = {'transform': 'translateX(0)'}  
   
 super().\_\_init\_\_('shake', keyframes, timing)  
  
class Pulse(Animation):  
 """Pulse animation"""  
 def \_\_init\_\_(self, scale: float = 1.1, duration: float = 1.0):  
 timing = AnimationTiming(duration=duration, iteration\_count='infinite')  
 super().\_\_init\_\_('pulse', {  
 '0%': {'transform': 'scale(1)'},  
 '50%': {'transform': f'scale({scale})'},  
 '100%': {'transform': 'scale(1)'}  
 }, timing)  
  
class AnimationManager:  
 """Animation management class"""  
   
 \_animations: Dict[str, Animation] = {}  
   
 @classmethod  
 def register(cls, animation: Animation):  
 """Register animation"""  
 cls.\_animations[animation.name] = animation  
   
 @classmethod  
 def get(cls, name: str) -> Optional[Animation]:  
 """Get registered animation"""  
 return cls.\_animations.get(name)  
   
 @classmethod  
 def get\_all\_css(cls) -> str:  
 """Get all animations CSS"""  
 return '\n\n'.join(anim.to\_css() for anim in cls.\_animations.values())  
   
 @classmethod  
 def create\_sequence(cls, \*animations: Tuple[str, float]) -> AnimationSequence:  
 """Create animation sequence from registered animations"""  
 sequence = []  
 for name, delay in animations:  
 animation = cls.get(name)  
 if animation:  
 sequence.append((animation, delay))  
 return AnimationSequence(\*sequence)  
  
# 预定义动画实例  
FADE\_IN = FadeIn()  
FADE\_OUT = FadeOut()  
SLIDE\_IN = Slide()  
SLIDE\_OUT = Slide('right')  
SLIDE\_UP = Slide('up')  
SLIDE\_DOWN = Slide('down')  
ROTATE = Rotate()  
SCALE = Scale()  
BOUNCE = Bounce()  
SHAKE = Shake()  
PULSE = Pulse()  
ELASTIC\_IN = Elastic('in')  
ELASTIC\_OUT = Elastic('out')  
SWING = Swing()  
WOBBLE = Wobble()  
ZOOM\_IN = Scale(0, 1, 0.3) # 从0缩放到1  
ZOOM\_OUT = Scale(1, 0, 0.3) # 从1缩放到0  
  
# 注册预定义动画  
for animation in [  
 FADE\_IN, FADE\_OUT,  
 SLIDE\_IN, SLIDE\_OUT, SLIDE\_UP, SLIDE\_DOWN,  
 ROTATE, SCALE, BOUNCE, SHAKE, PULSE,  
 ELASTIC\_IN, ELASTIC\_OUT, SWING, WOBBLE,  
 ZOOM\_IN, ZOOM\_OUT  
]:  
 AnimationManager.register(animation)

app.py

"""  
PytoWeb应用主类  
"""  
  
from \_\_future\_\_ import annotations  
from typing import Optional, Any, Callable, List, Dict, Union, TypeVar, TYPE\_CHECKING  
from dataclasses import dataclass  
from .server import Server  
from .router import Router  
from .components import Component  
from .vdom import VDOMRenderer  
import logging  
import sys  
import traceback  
from http import HTTPStatus  
  
if TYPE\_CHECKING:  
 from .middleware import Middleware  
  
T = TypeVar('T', bound='App')  
  
class AppError(Exception):  
 """PytoWeb应用异常基类"""  
 pass  
  
@dataclass  
class AppConfig:  
 """应用配置"""  
 host: str = "localhost"  
 port: int = 8000  
 debug: bool = False  
 static\_dir: str = "static"  
 template\_dir: str = "templates"  
 secret\_key: Optional[str] = None  
  
class App:  
 """PytoWeb应用主类"""  
   
 def \_\_init\_\_(self, config: Optional[AppConfig] = None):  
 """初始化应用"""  
 try:  
 self.config = config or AppConfig()  
 self.server = Server(self.config.host, self.config.port)  
 self.router = Router()  
 self.root: Optional[Component] = None  
 self.renderer = VDOMRenderer()  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
   
 # 配置日志  
 if self.config.debug:  
 logging.basicConfig(  
 level=logging.DEBUG,  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
 )  
   
 # 设置静态文件目录  
 self.server.static\_dir = self.config.static\_dir  
   
 # 注册默认路由处理器  
 self.router.add('/', self.\_handle\_root)  
 self.server.add\_route('/', self.router.dispatch)  
   
 except Exception as e:  
 raise AppError(f"Failed to initialize application: {e}") from e  
   
 def \_handle\_root(self, request: Dict[str, Any]) -> str:  
 """处理根路由请求"""  
 print("[DEBUG] Handling root request")  
 if self.root is None:  
 raise AppError("No root component mounted")  
 try:  
 html = self.render(self.root)  
 print(f"[DEBUG] Generated HTML length: {len(html)}")  
 return html  
 except Exception as e:  
 print(f"[DEBUG] Error rendering root: {e}")  
 raise AppError(f"Failed to render root: {e}") from e  
   
 def mount(self: T, component: Component) -> T:  
 """挂载根组件"""  
 try:  
 if not isinstance(component, Component):  
 raise AppError("Component must be an instance of Component")  
 self.root = component  
 return self  
 except Exception as e:  
 if isinstance(e, AppError):  
 raise  
 raise AppError(f"Failed to mount component: {e}") from e  
   
 def render(self, component: Component) -> str:  
 """渲染组件"""  
 try:  
 if not isinstance(component, Component):  
 raise AppError("Component must be an instance of Component")  
   
 vdom = component.render()  
 html = self.renderer.render\_to\_string(vdom)  
   
 return f"""  
 <!DOCTYPE html>  
 <html lang="en">  
 <head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>PytoWeb App</title>  
 <style>{self.\_get\_styles()}</style>  
 <script>{self.\_get\_scripts()}</script>  
 </head>  
 <body>  
 <div id="app">{html}</div>  
 </body>  
 </html>  
 """  
 except Exception as e:  
 if isinstance(e, AppError):  
 raise  
 raise AppError(f"Failed to render component: {e}") from e  
   
 def \_get\_styles(self) -> str:  
 """获取应用样式"""  
 try:  
 from .styles import get\_global\_styles  
 return get\_global\_styles()  
 except Exception as e:  
 self.\_logger.error(f"Failed to get styles: {e}")  
 return ""  
   
 def \_get\_scripts(self) -> str:  
 """获取应用脚本"""  
 try:  
 from .events import get\_client\_script  
 return get\_client\_script()  
 except Exception as e:  
 self.\_logger.error(f"Failed to get scripts: {e}")  
 return ""  
   
 def run(self, host: str = "127.0.0.1", port: int = 8000, debug: bool = False):  
 """运行应用"""  
 try:  
 if debug:  
 self.\_logger.setLevel(logging.DEBUG)  
 self.server.run(host, port)  
 except Exception as e:  
 raise AppError(f"Failed to run application: {e}") from e

components.py

"""  
PytoWeb组件系统  
  
提供基础和高级UI组件，支持虚拟滚动、拖放等功能。  
"""  
  
from \_\_future\_\_ import annotations  
from typing import (  
 Dict, Any, Optional, Callable, List, Set,  
 TypeVar, TypedDict, Union, TYPE\_CHECKING  
)  
from collections import OrderedDict  
import weakref  
import logging  
from .elements import Element  
from .styles import Style  
from .events import EventDelegate, Event  
import time  
import sys  
import asyncio  
import uuid  
import traceback  
from dataclasses import dataclass  
from datetime import datetime  
import json  
from functools import wraps  
  
if TYPE\_CHECKING:  
 from typing import Literal  
  
# 配置日志  
logging.basicConfig(level=logging.DEBUG)  
logger = logging.getLogger(\_\_name\_\_)  
  
# 类型别名  
T = TypeVar('T')  
OptionsType = List[Dict[str, str]]  
EventHandler = Callable[..., None]  
ComponentList = List['Component']  
PropDict = Dict[str, Any]  
StateDict = Dict[str, Any]  
  
class ComponentCache:  
 """组件缓存系统"""  
 \_instance = None  
   
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
   
 def \_\_init\_\_(self):  
 if not hasattr(self, 'initialized'):  
 self.\_cache: OrderedDict[str, tuple[Any, float]] = OrderedDict()  
 self.\_max\_size = 100 # 最大缓存项数  
 self.\_max\_memory = 100 \* 1024 \* 1024 # 最大内存使用(100MB)  
 self.\_ttl = 300 # 缓存过期时间(秒)  
 self.\_current\_memory = 0  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
 self.initialized = True  
   
 def get(self, key: str) -> Optional[Any]:  
 """获取缓存的组件"""  
 try:  
 if key in self.\_cache:  
 value, timestamp = self.\_cache[key]  
 current\_time = time.time()  
   
 # 检查是否过期  
 if current\_time - timestamp > self.\_ttl:  
 self.\_cache.pop(key)  
 self.\_current\_memory -= sys.getsizeof(value)  
 return None  
   
 # 更新访问顺序和时间戳  
 self.\_cache.move\_to\_end(key)  
 self.\_cache[key] = (value, current\_time)  
 return value  
 except Exception as e:  
 self.\_logger.error(f"Error getting cached component: {e}", exc\_info=True)  
 return None  
   
 def set(self, key: str, value: Any):  
 """缓存组件"""  
 try:  
 current\_time = time.time()  
 value\_size = sys.getsizeof(value)  
   
 # 检查单个值是否超过最大内存限制  
 if value\_size > self.\_max\_memory:  
 self.\_logger.warning(f"Value too large to cache: {value\_size} bytes")  
 return  
   
 # 如果已存在，先移除旧值  
 if key in self.\_cache:  
 old\_value, \_ = self.\_cache.pop(key)  
 self.\_current\_memory -= sys.getsizeof(old\_value)  
   
 # 清理过期和超出内存限制的缓存  
 while self.\_cache and (  
 len(self.\_cache) >= self.\_max\_size or  
 self.\_current\_memory + value\_size > self.\_max\_memory or  
 current\_time - next(iter(self.\_cache.values()))[1] > self.\_ttl  
 ):  
 removed\_key = next(iter(self.\_cache))  
 removed\_value, \_ = self.\_cache.pop(removed\_key)  
 self.\_current\_memory -= sys.getsizeof(removed\_value)  
   
 # 添加新值  
 self.\_cache[key] = (value, current\_time)  
 self.\_current\_memory += value\_size  
   
 except Exception as e:  
 self.\_logger.error(f"Error caching component: {e}", exc\_info=True)  
   
 def clear(self):  
 """清除缓存"""  
 self.\_cache.clear()  
 self.\_current\_memory = 0  
   
 def get\_stats(self) -> dict:  
 """获取缓存统计信息"""  
 return {  
 'size': len(self.\_cache),  
 'memory\_usage': self.\_current\_memory,  
 'max\_size': self.\_max\_size,  
 'max\_memory': self.\_max\_memory,  
 'ttl': self.\_ttl  
 }  
  
class Component:  
 """所有组件的基类"""  
   
 def \_\_init\_\_(self):  
 self.props: PropDict = {}  
 self.state: StateDict = {}  
 self.children: ComponentList = []  
 self.parent: Optional['Component'] = None  
 self.style = Style()  
 self.tag\_name = "div" # 默认标签  
 self.\_cache = ComponentCache()  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
 self.\_mounted = False  
 self.\_destroyed = False  
   
 # 生命周期事件  
 self.on\_before\_mount = EventDelegate()  
 self.on\_mounted = EventDelegate()  
 self.on\_before\_update = EventDelegate()  
 self.on\_updated = EventDelegate()  
 self.on\_before\_destroy = EventDelegate()  
 self.on\_destroyed = EventDelegate()  
 self.on\_error = EventDelegate()  
   
 # 状态变更事件  
 self.on\_state\_change = EventDelegate()  
 self.on\_prop\_change = EventDelegate()  
   
 self.\_memo\_cache = {}  
 self.\_memo\_deps = {}  
   
 self.\_lazy\_loaded = False  
 self.\_lazy\_loading = False  
 self.\_lazy\_error = None  
 self.\_lazy\_promise = None  
   
 def set\_prop(self, key: str, value: Any):  
 """设置属性"""  
 try:  
 old\_value = self.props.get(key)  
 if old\_value != value:  
 self.props[key] = value  
 self.on\_prop\_change(self, key, old\_value, value)  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error setting prop {key}: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def set\_state(self, key: str, value: Any):  
 """设置状态"""  
 try:  
 old\_value = self.state.get(key)  
 if old\_value != value:  
 self.state[key] = value  
 self.on\_state\_change(self, key, old\_value, value)  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error setting state {key}: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def add\_child(self, child: 'Component'):  
 """添加子组件"""  
 try:  
 child.parent = self  
 self.children.append(child)  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error adding child: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def remove\_child(self, child: 'Component'):  
 """移除子组件"""  
 try:  
 if child in self.children:  
 child.parent = None  
 self.children.remove(child)  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error removing child: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def mount(self):  
 """组件挂载"""  
 try:  
 if not self.\_mounted:  
 self.on\_before\_mount(self)  
 self.\_mounted = True  
 for child in self.children:  
 child.mount()  
 self.on\_mounted(self)  
 except Exception as e:  
 self.\_logger.error(f"Error mounting component: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def unmount(self):  
 """组件卸载"""  
 try:  
 if self.\_mounted and not self.\_destroyed:  
 self.on\_before\_destroy(self)  
 self.\_mounted = False  
 self.\_destroyed = True  
 for child in self.children:  
 child.unmount()  
 self.on\_destroyed(self)  
 except Exception as e:  
 self.\_logger.error(f"Error unmounting component: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def \_update(self):  
 """更新组件"""  
 try:  
 if self.\_mounted and not self.\_destroyed:  
 self.on\_before\_update(self)  
 # 实际更新逻辑  
 self.on\_updated(self)  
 except Exception as e:  
 self.\_logger.error(f"Error updating component: {e}", exc\_info=True)  
 self.on\_error(self, e)  
   
 def validate\_props(self, prop\_types: Dict[str, type]):  
 """验证属性类型"""  
 for key, expected\_type in prop\_types.items():  
 if key in self.props:  
 value = self.props[key]  
 if not isinstance(value, expected\_type):  
 raise TypeError(f"Prop '{key}' expected type {expected\_type.\_\_name\_\_}, got {type(value).\_\_name\_\_}")  
   
 def validate\_state(self, state\_types: Dict[str, type]):  
 """验证状态类型"""  
 for key, expected\_type in state\_types.items():  
 if key in self.state:  
 value = self.state[key]  
 if not isinstance(value, expected\_type):  
 raise TypeError(f"State '{key}' expected type {expected\_type.\_\_name\_\_}, got {type(value).\_\_name\_\_}")  
   
 def render(self):  
 """渲染组件"""  
 try:  
 print(f"[DEBUG] Rendering component: {self.\_\_class\_\_.\_\_name\_\_}")  
 element = Element(self.tag\_name)  
   
 # 添加样式  
 if self.style:  
 element.style.update(self.style.get\_all())  
 print(f"[DEBUG] Added styles: {self.style.get\_all()}")  
   
 # 添加子组件  
 for child in self.children:  
 try:  
 child\_element = child.render()  
 if child\_element:  
 element.add(child\_element)  
 print(f"[DEBUG] Added child element: {child.\_\_class\_\_.\_\_name\_\_}")  
 else:  
 print(f"[WARNING] Child {child.\_\_class\_\_.\_\_name\_\_} rendered None")  
 except Exception as e:  
 print(f"[ERROR] Failed to render child {child.\_\_class\_\_.\_\_name\_\_}: {e}")  
 raise  
   
 return element  
 except Exception as e:  
 print(f"[ERROR] Failed to render {self.\_\_class\_\_.\_\_name\_\_}: {e}")  
 raise  
   
 def memo(self, key: str, fn: Callable[..., Any], \*deps: Any) -> Any:  
 """记忆化计算结果  
   
 Args:  
 key: 缓存键名  
 fn: 要记忆化的函数  
 deps: 依赖项，当这些值变化时重新计算  
   
 Returns:  
 记忆化的计算结果  
 """  
 current\_deps = tuple(deps)  
   
 # 检查依赖是否变化  
 if (key not in self.\_memo\_cache or  
 key not in self.\_memo\_deps or  
 self.\_memo\_deps[key] != current\_deps):  
   
 # 重新计算并缓存结果  
 self.\_memo\_cache[key] = fn()  
 self.\_memo\_deps[key] = current\_deps  
   
 return self.\_memo\_cache[key]  
   
 def clear\_memo(self, key: Optional[str] = None):  
 """清除记忆化缓存  
   
 Args:  
 key: 要清除的特定缓存键,如果为None则清除所有缓存  
 """  
 if key is None:  
 self.\_memo\_cache.clear()  
 self.\_memo\_deps.clear()  
 else:  
 self.\_memo\_cache.pop(key, None)  
 self.\_memo\_deps.pop(key, None)  
  
 def lazy\_load(self, loader: Callable[[], Awaitable[Any]]) -> None:  
 """懒加载组件内容  
   
 Args:  
 loader: 异步加载函数  
 """  
 if not self.\_lazy\_loaded and not self.\_lazy\_loading:  
 self.\_lazy\_loading = True  
 self.\_lazy\_promise = asyncio.create\_task(self.\_do\_lazy\_load(loader))  
   
 async def \_do\_lazy\_load(self, loader: Callable[[], Awaitable[Any]]) -> None:  
 """执行懒加载  
   
 Args:  
 loader: 异步加载函数  
 """  
 try:  
 result = await loader()  
 self.\_handle\_lazy\_load\_success(result)  
 except Exception as e:  
 self.\_handle\_lazy\_load\_error(e)  
   
 def \_handle\_lazy\_load\_success(self, result: Any) -> None:  
 """处理懒加载成功  
   
 Args:  
 result: 加载结果  
 """  
 self.\_lazy\_loaded = True  
 self.\_lazy\_loading = False  
 self.\_lazy\_error = None  
 self.state['lazy\_result'] = result  
 self.\_update()  
   
 def \_handle\_lazy\_load\_error(self, error: Exception) -> None:  
 """处理懒加载错误  
   
 Args:  
 error: 错误信息  
 """  
 self.\_lazy\_loaded = False  
 self.\_lazy\_loading = False  
 self.\_lazy\_error = error  
 self.\_update()  
   
 def is\_lazy\_loaded(self) -> bool:  
 """检查是否已完成懒加载"""  
 return self.\_lazy\_loaded  
   
 def is\_lazy\_loading(self) -> bool:  
 """检查是否正在懒加载"""  
 return self.\_lazy\_loading  
   
 def get\_lazy\_error(self) -> Optional[Exception]:  
 """获取懒加载错误信息"""  
 return self.\_lazy\_error  
  
class AsyncComponentMixin:  
 """为组件添加异步支持的Mixin类"""  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.\_cache = ComponentCache()  
 self.\_pending\_updates = {}  
   
 async def update\_async(self, \*\*kwargs):  
 """异步更新组件状态"""  
 update\_id = str(uuid.uuid4())  
 self.\_pending\_updates[update\_id] = asyncio.Future()  
   
 try:  
 await self.on\_before\_update.emit\_async()  
 self.state.update(kwargs)  
 await self.on\_updated.emit\_async()  
 self.\_pending\_updates[update\_id].set\_result(True)  
 except Exception as e:  
 self.\_pending\_updates[update\_id].set\_exception(e)  
 finally:  
 del self.\_pending\_updates[update\_id]  
   
 async def render\_async(self):  
 """异步渲染组件"""  
 cache\_key = self.\_get\_cache\_key()  
 cached = self.\_cache.get(cache\_key)  
 if cached:  
 return cached  
   
 try:  
 await self.on\_before\_mount.emit\_async()  
 result = await self.\_render\_async\_impl()  
 await self.on\_mounted.emit\_async()  
   
 self.\_cache.set(cache\_key, result)  
 return result  
 except Exception as e:  
 self.logger.error(f"Error in async rendering: {e}")  
 raise  
   
 async def \_render\_async\_impl(self):  
 """异步渲染实现"""  
 raise NotImplementedError("Async components must implement \_render\_async\_impl")  
  
class AsyncComponent(AsyncComponentMixin, Component):  
 """异步组件基类"""  
 pass  
  
class Suspense(Component):  
 """处理异步加载状态的组件"""  
 def \_\_init\_\_(self,  
 component: AsyncComponent,  
 fallback: Optional[Component] = None,  
 error\_fallback: Optional[Component] = None):  
 super().\_\_init\_\_()  
 self.set\_prop('component', component)  
 self.set\_prop('fallback', fallback or self.\_default\_fallback())  
 self.set\_prop('error\_fallback', error\_fallback or self.\_default\_error())  
   
 self.state.update({  
 'loading': True,  
 'error': None  
 })  
   
 def \_default\_fallback(self):  
 """默认加载组件"""  
 loading = Component()  
 loading.tag\_name = "div"  
 loading.style.add(  
 text\_align="center",  
 padding="1rem"  
 )  
 loading.set\_text("Loading...")  
 return loading  
   
 def \_default\_error(self):  
 """默认错误组件"""  
 error = Component()  
 error.tag\_name = "div"  
 error.style.add(  
 color="red",  
 text\_align="center",  
 padding="1rem"  
 )  
 error.set\_text("An error occurred")  
 return error  
   
 async def render\_async(self):  
 """异步渲染"""  
 try:  
 if self.state['loading']:  
 return self.props['fallback']  
   
 result = await self.props['component'].render\_async()  
 self.state['loading'] = False  
 return result  
 except Exception as e:  
 self.state['error'] = str(e)  
 self.logger.error(f"Error in Suspense: {e}")  
 return self.props['error\_fallback']  
  
class ErrorBoundary(Component):  
 """错误边界组件，用于捕获和处理子组件中的错误"""  
   
 def \_\_init\_\_(self,  
 children: list[Component],  
 fallback: Optional[Callable[[Exception], Component]] = None):  
 super().\_\_init\_\_()  
 self.set\_prop('children', children)  
 self.set\_prop('fallback', fallback or self.\_default\_fallback)  
   
 self.state.update({  
 'error': None,  
 'error\_info': None  
 })  
   
 self.\_error\_handler = ErrorHandler.get\_instance()  
   
 def \_default\_fallback(self, error: Exception) -> Component:  
 """默认错误回退组件"""  
 error\_component = Component()  
 error\_component.tag\_name = "div"  
 error\_component.style.add(  
 color="red",  
 padding="1rem",  
 border="1px solid red",  
 margin="1rem",  
 background\_color="rgba(255,0,0,0.1)"  
 )  
 error\_component.set\_text(f"Error: {str(error)}")  
 return error\_component  
   
 def render(self):  
 """渲染错误边界"""  
 if self.state['error']:  
 error\_component = self.props['fallback'](self.state['error'])  
 return error\_component  
   
 try:  
 return self.props['children']  
 except Exception as e:  
 self.state['error'] = e  
 self.state['error\_info'] = self.\_error\_handler.\_get\_error\_context()  
 self.\_error\_handler.handle\_error(e, self.state['error\_info'])  
 return self.props['fallback'](e)  
  
@dataclass  
class ErrorContext:  
 """错误上下文信息"""  
 component: Optional[str] = None  
 function: Optional[str] = None  
 line\_number: Optional[int] = None  
 file\_path: Optional[str] = None  
 stack\_trace: Optional[str] = None  
 additional\_info: Dict[str, Any] = None  
  
@dataclass  
class ErrorReport:  
 """详细错误报告"""  
 error\_type: str  
 message: str  
 context: ErrorContext  
 timestamp: datetime  
 severity: str  
 handled: bool  
  
class ErrorHandler:  
 """中央错误处理系统"""  
 \_instance = None  
   
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
   
 def \_\_init\_\_(self):  
 if not hasattr(self, 'initialized'):  
 self.error\_listeners: List[Callable[[ErrorReport], None]] = []  
 self.error\_history: List[ErrorReport] = []  
 self.max\_history = 100  
 self.logger = logging.getLogger('pytoweb.errors')  
 self.initialized = True  
   
 @classmethod  
 def get\_instance(cls):  
 return cls()  
   
 def add\_listener(self, listener: Callable[[ErrorReport], None]):  
 """添加错误监听器"""  
 self.error\_listeners.append(listener)  
   
 def remove\_listener(self, listener: Callable[[ErrorReport], None]):  
 """移除错误监听器"""  
 self.error\_listeners.remove(listener)  
   
 def handle\_error(self, error: Exception, context: Optional[ErrorContext] = None):  
 """处理错误"""  
 if context is None:  
 context = self.\_get\_error\_context()  
   
 report = ErrorReport(  
 error\_type=type(error).\_\_name\_\_,  
 message=str(error),  
 context=context,  
 timestamp=datetime.now(),  
 severity=self.\_get\_error\_severity(error),  
 handled=True  
 )  
   
 self.error\_history.append(report)  
 if len(self.error\_history) > self.max\_history:  
 self.error\_history.pop(0)  
   
 for listener in self.error\_listeners:  
 try:  
 listener(report)  
 except Exception as e:  
 self.logger.error(f"Error in error listener: {e}")  
   
 self.logger.error(f"Error: {report.message}", exc\_info=True)  
   
 def \_get\_error\_context(self) -> ErrorContext:  
 """从当前异常获取上下文"""  
 tb = sys.exc\_info()[2]  
 while tb.tb\_next:  
 tb = tb.tb\_next  
   
 frame = tb.tb\_frame  
 return ErrorContext(  
 function=frame.f\_code.co\_name,  
 line\_number=tb.tb\_lineno,  
 file\_path=frame.f\_code.co\_filename,  
 stack\_trace=traceback.format\_exc()  
 )  
   
 def \_get\_error\_severity(self, error: Exception) -> str:  
 """确定错误严重性"""  
 if isinstance(error, (SystemError, MemoryError)):  
 return "CRITICAL"  
 if isinstance(error, (ValueError, TypeError)):  
 return "ERROR"  
 return "WARNING"  
   
 def get\_error\_summary(self) -> Dict[str, Any]:  
 """获取最近错误的摘要"""  
 return {  
 'total\_errors': len(self.error\_history),  
 'error\_types': self.\_count\_error\_types(),  
 'recent\_errors': [  
 {  
 'type': e.error\_type,  
 'message': e.message,  
 'timestamp': e.timestamp.isoformat()  
 }  
 for e in self.error\_history[-5:]  
 ]  
 }  
   
 def \_count\_error\_types(self) -> Dict[str, int]:  
 """统计每种错误类型的出现次数"""  
 counts = {}  
 for error in self.error\_history:  
 counts[error.error\_type] = counts.get(error.error\_type, 0) + 1  
 return counts  
   
 def export\_error\_report(self, filepath: str):  
 """导出错误历史到文件"""  
 try:  
 with open(filepath, 'w') as f:  
 json.dump(  
 {  
 'error\_summary': self.get\_error\_summary(),  
 'full\_history': [  
 {  
 'type': e.error\_type,  
 'message': e.message,  
 'timestamp': e.timestamp.isoformat(),  
 'severity': e.severity,  
 'context': {  
 'component': e.context.component,  
 'function': e.context.function,  
 'line': e.context.line\_number,  
 'file': e.context.file\_path,  
 'stack\_trace': e.context.stack\_trace  
 }  
 }  
 for e in self.error\_history  
 ]  
 },  
 f,  
 indent=2  
 )  
 except Exception as e:  
 self.logger.error(f"Failed to export error report: {e}")  
  
def error\_boundary(fallback\_component: Optional[Callable[[Exception], Component]] = None):  
 """错误边界装饰器"""  
 def decorator(component\_class):  
 original\_render = component\_class.render  
   
 @wraps(original\_render)  
 def wrapped\_render(self, \*args, \*\*kwargs):  
 boundary = ErrorBoundary(  
 children=[original\_render(self, \*args, \*\*kwargs)],  
 fallback=fallback\_component  
 )  
 return boundary.render()  
   
 component\_class.render = wrapped\_render  
 return component\_class  
   
 return decorator  
  
class Button(Component):  
 """预构建的Button组件"""  
   
 def \_\_init\_\_(self, text: str, on\_click: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "button"  
 self.set\_prop('text', text)  
 if on\_click:  
 self.set\_prop('on\_click', on\_click)  
   
 def render(self) -> Element:  
 button = Element(self.tag\_name, text=self.props['text'])  
 if 'on\_click' in self.props:  
 button.on('click', self.props['on\_click'])  
 return button  
  
class Container(Component):  
 """预构建的Container组件"""  
   
 def \_\_init\_\_(self, \*children: Component):  
 super().\_\_init\_\_()  
 for child in children:  
 self.add\_child(child)  
   
 def render(self) -> Element:  
 container = Element(self.tag\_name)  
 for child in self.children:  
 container.add(child.render())  
 return container  
  
class Input(Component):  
 """预构建的Input组件"""  
   
 def \_\_init\_\_(self, placeholder: str = "", value: str = "", on\_change: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "input"  
 self.set\_prop('placeholder', placeholder)  
 self.set\_prop('value', value)  
 if on\_change:  
 self.set\_prop('on\_change', on\_change)  
   
 def render(self) -> Element:  
 input\_elem = Element(self.tag\_name)  
 input\_elem.set\_attr('placeholder', self.props['placeholder'])  
 input\_elem.set\_attr('value', self.props['value'])  
 if 'on\_change' in self.props:  
 input\_elem.on('change', self.props['on\_change'])  
 return input\_elem  
  
class Form(Component):  
 """预构建的Form组件"""  
   
 def \_\_init\_\_(self, on\_submit: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "form"  
 if on\_submit:  
 self.set\_prop('on\_submit', on\_submit)  
   
 def render(self) -> Element:  
 form = Element(self.tag\_name)  
 if 'on\_submit' in self.props:  
 form.on('submit', self.props['on\_submit'])  
 for child in self.children:  
 form.add(child.render())  
 return form  
  
class Text(Component):  
 """文本组件"""  
   
 def \_\_init\_\_(self, text: str, tag: str = "span"):  
 super().\_\_init\_\_()  
 self.tag\_name = tag  
 self.set\_prop('text', text)  
   
 def render(self) -> Element:  
 return Element(self.tag\_name, text=self.text)  
  
class Image(Component):  
 """图像组件"""  
   
 def \_\_init\_\_(self, src: str, alt: str = "", width: str = "", height: str = ""):  
 super().\_\_init\_\_()  
 self.tag\_name = "img"  
 self.set\_prop('src', src)  
 self.set\_prop('alt', alt)  
 if width:  
 self.set\_prop('width', width)  
 if height:  
 self.set\_prop('height', height)  
   
 def render(self) -> Element:  
 img = Element(self.tag\_name)  
 img.set\_attr('src', self.src)  
 img.set\_attr('alt', self.alt)  
 if 'width' in self.props:  
 img.set\_attr('width', self.width)  
 if 'height' in self.props:  
 img.set\_attr('height', self.height)  
 return img  
  
class Link(Component):  
 """链接组件"""  
   
 def \_\_init\_\_(self, href: str, text: str = "", target: str = "\_self"):  
 super().\_\_init\_\_()  
 self.tag\_name = "a"  
 self.set\_prop('href', href)  
 self.set\_prop('text', text)  
 self.set\_prop('target', target)  
   
 def render(self) -> Element:  
 link = Element(self.tag\_name, text=self.text)  
 link.set\_attr('href', self.href)  
 link.set\_attr('target', self.target)  
 return link  
  
class List(Component):  
 """列表组件"""  
   
 def \_\_init\_\_(self, items: list[str] | None = None, ordered: bool = False):  
 super().\_\_init\_\_()  
 self.tag\_name = "ol" if ordered else "ul"  
 self.set\_prop('items', items or [])  
  
 def add\_item(self, item: str):  
 if 'items' not in self.props:  
 self.props['items'] = []  
 self.props['items'].append(item)  
   
 def render(self) -> Element:  
 list\_elem = Element(self.tag\_name)  
 for item in self.props.get('items', []):  
 li = Element('li', text=str(item))  
 list\_elem.add(li)  
 return list\_elem  
  
class Card(Component):  
 """卡片组件"""  
   
 def \_\_init\_\_(self, title: str = "", body: str = "", footer: str = ""):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('title', title)  
 self.set\_prop('body', body)  
 self.set\_prop('footer', footer)  
   
 def render(self) -> Element:  
 card = Element(self.tag\_name)  
 card.add\_class('card')  
   
 if self.title:  
 header = Element('div')  
 header.add\_class('card-header')  
 header.add(Element('h3', text=self.title))  
 card.add(header)  
   
 body = Element('div')  
 body.add\_class('card-body')  
 body.add(Element('p', text=self.body))  
 card.add(body)  
   
 if self.footer:  
 footer = Element('div')  
 footer.add\_class('card-footer')  
 footer.add(Element('p', text=self.footer))  
 card.add(footer)  
   
 return card  
  
class Grid(Component):  
 """网格布局组件"""  
   
 def \_\_init\_\_(self, columns: int = 12, gap: str = "1rem"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('columns', columns)  
 self.set\_prop('gap', gap)  
 self.style.add(  
 display="grid",  
 grid\_template\_columns=f"repeat({columns}, 1fr)",  
 gap=gap  
 )  
   
 def add\_item(self, component: Component, column\_span: int = 1):  
 component.style.add(grid\_column=f"span {column\_span}")  
 self.add\_child(component)  
   
 def render(self) -> Element:  
 grid = Element(self.tag\_name)  
 for child in self.children:  
 grid.add(child.render())  
 return grid  
  
class Select(Component):  
 """选择组件"""  
   
 def \_\_init\_\_(self, options: OptionsType, value: str = "", on\_change: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "select"  
 self.set\_prop('options', options)  
 self.set\_prop('value', value)  
 if on\_change:  
 self.set\_prop('on\_change', on\_change)  
   
 def render(self) -> Element:  
 select = Element(self.tag\_name)  
 if 'on\_change' in self.props:  
 select.on('change', self.on\_change)  
   
 for option in self.options:  
 opt = Element('option')  
 opt.set\_attr('value', option.get('value', ''))  
 if option.get('value') == self.value:  
 opt.set\_attr('selected', 'selected')  
 opt.text = option.get('label', option.get('value', ''))  
 select.add(opt)  
   
 return select  
  
class Checkbox(Component):  
 """复选框组件"""  
   
 def \_\_init\_\_(self, label: str = "", checked: bool = False, on\_change: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "input"  
 self.set\_prop('type', 'checkbox')  
 self.set\_prop('label', label)  
 self.set\_prop('checked', checked)  
 if on\_change:  
 self.set\_prop('on\_change', on\_change)  
   
 def render(self) -> Element:  
 container = Element('div')  
   
 input\_elem = Element(self.tag\_name)  
 input\_elem.set\_attr('type', 'checkbox')  
 if self.checked:  
 input\_elem.set\_attr('checked', 'checked')  
 if 'on\_change' in self.props:  
 input\_elem.on('change', self.on\_change)  
 container.add(input\_elem)  
   
 if self.label:  
 label = Element('label')  
 label.text = self.label  
 container.add(label)  
   
 return container  
  
class Radio(Component):  
 """单选框组件"""  
   
 def \_\_init\_\_(self, name: str, value: str, label: str = "", checked: bool = False, on\_change: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "input"  
 self.set\_prop('type', 'radio')  
 self.set\_prop('name', name)  
 self.set\_prop('value', value)  
 self.set\_prop('label', label)  
 self.set\_prop('checked', checked)  
 if on\_change:  
 self.set\_prop('on\_change', on\_change)  
   
 def render(self) -> Element:  
 container = Element('div')  
   
 input\_elem = Element(self.tag\_name)  
 input\_elem.set\_attr('type', 'radio')  
 input\_elem.set\_attr('name', self.name)  
 input\_elem.set\_attr('value', self.value)  
 if self.checked:  
 input\_elem.set\_attr('checked', 'checked')  
 if 'on\_change' in self.props:  
 input\_elem.on('change', self.on\_change)  
 container.add(input\_elem)  
   
 if self.label:  
 label = Element('label')  
 label.text = self.label  
 container.add(label)  
   
 return container  
  
class TextArea(Component):  
 """文本域组件"""  
   
 def \_\_init\_\_(self, value: str = "", placeholder: str = "", rows: int = 3, on\_change: Optional[Callable] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "textarea"  
 self.set\_prop('value', value)  
 self.set\_prop('placeholder', placeholder)  
 self.set\_prop('rows', rows)  
 if on\_change:  
 self.set\_prop('on\_change', on\_change)  
   
 def render(self) -> Element:  
 textarea = Element(self.tag\_name, text=self.value)  
 textarea.set\_attr('placeholder', self.placeholder)  
 textarea.set\_attr('rows', str(self.rows))  
 if 'on\_change' in self.props:  
 textarea.on('change', self.on\_change)  
 return textarea  
  
class Navbar(Component):  
 """导航栏组件"""  
   
 def \_\_init\_\_(self, brand: str = "", items: list[dict[str, str]] = None, theme: str = "light"):  
 super().\_\_init\_\_()  
 self.tag\_name = "nav"  
 self.set\_prop('brand', brand)  
 self.set\_prop('items', items or [])  
 self.set\_prop('theme', theme)  
 self.style.add(  
 display="flex",  
 align\_items="center",  
 padding="1rem",  
 background\_color="#ffffff" if theme == "light" else "#343a40",  
 color="#000000" if theme == "light" else "#ffffff"  
 )  
   
 def add\_item(self, text: str, href: str = "#", active: bool = False):  
 self.props['items'].append({  
 'text': text,  
 'href': href,  
 'active': active  
 })  
   
 def render(self) -> Element:  
 nav = Element(self.tag\_name)  
   
 if self.brand:  
 brand = Element('a')  
 brand.add\_class('navbar-brand')  
 brand.set\_attr('href', '#')  
 brand.text = self.brand  
 brand.style.add(  
 font\_size="1.25rem",  
 padding\_right="1rem",  
 text\_decoration="none",  
 color="inherit"  
 )  
 nav.add(brand)  
   
 items\_container = Element('div')  
 items\_container.add\_class('navbar-items')  
 items\_container.style.add(  
 display="flex",  
 gap="1rem"  
 )  
   
 for item in self.items:  
 link = Element('a')  
 link.set\_attr('href', item.get('href', '#'))  
 link.text = item.get('text', '')  
 link.style.add(  
 text\_decoration="none",  
 color="inherit"  
 )  
 if item.get('active'):  
 link.style.add(font\_weight="bold")  
 items\_container.add(link)  
   
 nav.add(items\_container)  
 return nav  
  
class Flex(Component):  
 """Flexbox容器组件"""  
   
 def \_\_init\_\_(self, direction: str = "row", justify: str = "flex-start", align: str = "stretch", wrap: bool = False, gap: str = "0"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.style.add(  
 display="flex",  
 flex\_direction=direction,  
 justify\_content=justify,  
 align\_items=align,  
 flex\_wrap="wrap" if wrap else "nowrap",  
 gap=gap  
 )  
   
 def render(self) -> Element:  
 flex = Element(self.tag\_name)  
 for child in self.children:  
 flex.add(child.render())  
 return flex  
  
class ModernModal(Component):  
 """现代模态对话框组件"""  
 def \_\_init\_\_(self,  
 title: str,  
 content: str,  
 size: Literal["sm", "md", "lg", "xl"] = "md",  
 centered: bool = True,  
 closable: bool = True):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('title', title)  
 self.set\_prop('content', content)  
 self.set\_prop('size', size)  
 self.set\_prop('centered', centered)  
 self.set\_prop('closable', closable)  
   
 self.state.update({  
 'visible': False  
 })  
   
 # 设置样式  
 self.style.add(  
 position="fixed",  
 top="0",  
 left="0",  
 width="100%",  
 height="100%",  
 display="flex",  
 align\_items="center" if centered else "flex-start",  
 justify\_content="center",  
 background\_color="rgba(0, 0, 0, 0.5)",  
 z\_index="1000",  
 opacity="0",  
 visibility="hidden",  
 transition="opacity 0.3s ease-in-out, visibility 0.3s ease-in-out"  
 )  
   
 def show(self) -> None:  
 """显示模态对话框"""  
 self.set\_state('visible', True)  
 self.style.add(  
 opacity="1",  
 visibility="visible"  
 )  
   
 def hide(self) -> None:  
 """隐藏模态对话框"""  
 self.set\_state('visible', False)  
 self.style.add(  
 opacity="0",  
 visibility="hidden"  
 )  
   
 def \_get\_size\_width(self) -> str:  
 """Get modal width based on size"""  
 size\_map = {  
 'sm': '300px',  
 'md': '500px',  
 'lg': '800px',  
 'xl': '1140px'  
 }  
 return size\_map.get(self.props['size'], '500px')  
   
 def render(self):  
 """渲染模态对话框"""  
 dialog = Component()  
 dialog.tag\_name = "div"  
 dialog.style.add(  
 background\_color="#ffffff",  
 border\_radius="0.5rem",  
 box\_shadow="0 25px 50px -12px rgba(0, 0, 0, 0.25)",  
 max\_width=self.\_get\_size\_width(),  
 width="100%",  
 max\_height="90vh",  
 display="flex",  
 flex\_direction="column",  
 transform=f"scale({1 if self.state['visible'] else 0.9})",  
 transition="transform 0.3s ease-in-out"  
 )  
   
 # Header  
 header = Component()  
 header.tag\_name = "div"  
 header.style.add(  
 padding="1rem",  
 border\_bottom="1px solid #e5e7eb",  
 display="flex",  
 align\_items="center",  
 justify\_content="space-between"  
 )  
   
 title = Component()  
 title.tag\_name = "h3"  
 title.style.add(  
 margin="0",  
 font\_size="1.25rem",  
 font\_weight="600",  
 color="#111827"  
 )  
 title.set\_text(self.props['title'])  
 header.add\_child(title)  
   
 if self.props['closable']:  
 close\_button = Component()  
 close\_button.tag\_name = "button"  
 close\_button.style.add(  
 background="none",  
 border="none",  
 padding="0.5rem",  
 cursor="pointer",  
 color="#6b7280"  
 )  
 close\_button.set\_text("×")  
 close\_button.on\_click.add(self.hide)  
 header.add\_child(close\_button)  
   
 dialog.add\_child(header)  
   
 # Content  
 content = Component()  
 content.tag\_name = "div"  
 content.style.add(  
 padding="1rem",  
 overflow\_y="auto"  
 )  
   
 if isinstance(self.props['content'], str):  
 content.set\_text(self.props['content'])  
 else:  
 content.add\_child(self.props['content'])  
   
 dialog.add\_child(content)  
   
 return dialog  
  
class ModernToast(Component):  
 """现代吐司通知组件"""  
   
 def \_\_init\_\_(self,  
 message: str,  
 type: str = "info",  
 duration: int = 3000,  
 position: str = "bottom-right"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('message', message)  
 self.set\_prop('type', type)  
 self.set\_prop('duration', duration)  
 self.set\_prop('position', position)  
   
 self.state.update({  
 'visible': False  
 })  
   
 # 设置样式  
 self.style.add(  
 position="fixed",  
 padding="1rem",  
 border\_radius="0.5rem",  
 background\_color=self.\_get\_background\_color(),  
 color="#ffffff",  
 box\_shadow="0 10px 15px -3px rgba(0, 0, 0, 0.1)",  
 max\_width="24rem",  
 opacity="0",  
 transform="translateY(1rem)",  
 transition="opacity 0.3s ease-in-out, transform 0.3s ease-in-out",  
 \*\*self.\_get\_position\_style()  
 )  
   
 def show(self):  
 """显示吐司通知"""  
 self.set\_state('visible', True)  
 self.style.add(  
 opacity="1",  
 transform="translateY(0)"  
 )  
   
 # Auto hide  
 if self.props['duration'] > 0:  
 def hide():  
 self.hide()  
 setTimeout(hide, self.props['duration'])  
   
 def hide(self):  
 """隐藏吐司通知"""  
 self.set\_state('visible', False)  
 self.style.add(  
 opacity="0",  
 transform="translateY(1rem)"  
 )  
   
 def \_get\_background\_color(self) -> str:  
 """Get background color based on type"""  
 colors = {  
 "info": "#3b82f6",  
 "success": "#10b981",  
 "warning": "#f59e0b",  
 "error": "#ef4444"  
 }  
 return colors.get(self.props['type'], colors['info'])  
   
 def \_get\_position\_style(self) -> dict[str, str]:  
 """Get position style"""  
 positions = {  
 "top-left": {"top": "1rem", "left": "1rem"},  
 "top-right": {"top": "1rem", "right": "1rem"},  
 "bottom-left": {"bottom": "1rem", "left": "1rem"},  
 "bottom-right": {"bottom": "1rem", "right": "1rem"}  
 }  
 return positions.get(self.props['position'], positions['bottom-right'])  
   
 def render(self):  
 """Render toast"""  
 container = Component()  
 container.tag\_name = "div"  
 container.style.add(  
 display="flex",  
 align\_items="center",  
 gap="0.5rem"  
 )  
   
 # Icon  
 icon = Component()  
 icon.tag\_name = "span"  
 icon.style.add(  
 font\_size="1.25rem"  
 )  
 icon.set\_text(self.\_get\_icon())  
 container.add\_child(icon)  
   
 # Message  
 message = Component()  
 message.tag\_name = "span"  
 message.set\_text(self.props['message'])  
 container.add\_child(message)  
   
 return container  
   
 def \_get\_icon(self) -> str:  
 """Get icon based on type"""  
 icons = {  
 "info": "ℹ",  
 "success": "✓",  
 "warning": "⚠",  
 "error": "✕"  
 }  
 return icons.get(self.props['type'], icons['info'])  
  
class ModernTabs(Component):  
 """现代选项卡组件"""  
   
 def \_\_init\_\_(self,  
 tabs: list[dict[str, Any]],  
 active\_index: int = 0,  
 variant: str = "default"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('tabs', tabs)  
 self.set\_prop('variant', variant)  
   
 self.state.update({  
 'active\_index': active\_index  
 })  
  
 def \_handle\_tab\_click(self, index: int):  
 """Handle tab click"""  
 self.set\_state('active\_index', index)  
   
 def render(self):  
 """Render tabs"""  
 container = Component()  
 container.tag\_name = "div"  
   
 # Tab list  
 tab\_list = Component()  
 tab\_list.tag\_name = "div"  
 tab\_list.style.add(  
 display="flex",  
 border\_bottom="1px solid #e5e7eb"  
 )  
   
 for i, tab in enumerate(self.props['tabs']):  
 tab\_button = Component()  
 tab\_button.tag\_name = "button"  
 tab\_button.style.add(  
 padding="0.75rem 1rem",  
 border="none",  
 background="none",  
 font\_weight="500",  
 color="#6b7280" if i != self.state['active\_index'] else "#111827",  
 border\_bottom=f"2px solid {'transparent' if i != self.state['active\_index'] else '#3b82f6'}",  
 cursor="pointer",  
 transition="all 0.2s ease-in-out"  
 )  
 tab\_button.set\_text(tab['label'])  
 tab\_button.on\_click.add(lambda e, i=i: self.\_handle\_tab\_click(i))  
 tab\_list.add\_child(tab\_button)  
   
 container.add\_child(tab\_list)  
   
 # Tab panels  
 panel\_container = Component()  
 panel\_container.tag\_name = "div"  
 panel\_container.style.add(  
 padding="1rem"  
 )  
   
 active\_tab = self.props['tabs'][self.state['active\_index']]  
 if isinstance(active\_tab['content'], str):  
 panel\_container.set\_text(active\_tab['content'])  
 else:  
 panel\_container.add\_child(active\_tab['content'])  
   
 container.add\_child(panel\_container)  
   
 return container  
  
class ModernAccordion(Component):  
 """现代手风琴组件"""  
   
 def \_\_init\_\_(self,  
 items: list[dict[str, Any]],  
 multiple: bool = False):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('items', items)  
 self.set\_prop('multiple', multiple)  
   
 self.state.update({  
 'expanded': set()  
 })  
  
 def \_toggle\_item(self, index: int):  
 """Toggle accordion item"""  
 expanded = self.state['expanded'].copy()  
   
 if not self.props['multiple']:  
 expanded.clear()  
   
 if index in expanded:  
 expanded.remove(index)  
 else:  
 expanded.add(index)  
   
 self.set\_state('expanded', expanded)  
   
 def render(self):  
 """Render accordion"""  
 container = Component()  
 container.tag\_name = "div"  
 container.style.add(  
 border="1px solid #e5e7eb",  
 border\_radius="0.5rem",  
 overflow="hidden"  
 )  
   
 for i, item in enumerate(self.props['items']):  
 # Item container  
 item\_container = Component()  
 item\_container.tag\_name = "div"  
 item\_container.style.add(  
 border\_top="1px solid #e5e7eb" if i > 0 else "none"  
 )  
   
 # Header  
 header = Component()  
 header.tag\_name = "button"  
 header.style.add(  
 width="100%",  
 padding="1rem",  
 background="none",  
 border="none",  
 text\_align="left",  
 cursor="pointer",  
 display="flex",  
 align\_items="center",  
 justify\_content="space-between"  
 )  
   
 # Expand/collapse icon  
 has\_children = 'children' in item and item['children']  
 if has\_children:  
 icon = Component()  
 icon.tag\_name = "span"  
 icon.style.add(  
 margin\_right="0.5rem",  
 transition="transform 0.2s"  
 )  
 if i in self.state['expanded']:  
 icon.style.add(transform="rotate(90deg)")  
 icon.add(Element('span', text="▶"))  
 header.add(icon)  
   
 # Node icon (if provided)  
 if 'icon' in item:  
 node\_icon = Component()  
 node\_icon.tag\_name = "span"  
 node\_icon.style.add(margin\_right="0.5rem")  
 node\_icon.add(Element('span', text=item['icon']))  
 header.add(node\_icon)  
   
 # Node label  
 label = Component()  
 label.tag\_name = "span"  
 label.add(Element('span', text=item['label']))  
 header.add(label)  
   
 # Add click handler for expansion toggle  
 if has\_children:  
 header.on('click', lambda: self.\_toggle\_item(i))  
   
 item\_container.add(header)  
   
 # Render children if node is expanded  
 if has\_children and i in self.state['expanded']:  
 children\_container = Component()  
 for child in item['children']:  
 children\_container.add(self.\_render\_node(child, 1))  
 item\_container.add(children\_container)  
   
 container.add\_child(item\_container)  
   
 return container  
  
 def \_render\_node(self, node: Dict[str, Any], level: int = 0) -> Element:  
 """Render a single node and its children"""  
 node\_container = Element('div')  
   
 # Node header  
 header = Element('div')  
 header.style.add(  
 display="flex",  
 align\_items="center",  
 padding="0.5rem",  
 padding\_left=f"{level \* 1.5 + 0.5}rem",  
 cursor="pointer",  
 transition="background-color 0.2s"  
 )  
 header.add\_hover\_style(background\_color="#f5f5f5")  
   
 # Expand/collapse icon  
 has\_children = 'children' in node and node['children']  
 if has\_children:  
 icon = Element('span')  
 icon.style.add(  
 margin\_right="0.5rem",  
 transition="transform 0.2s"  
 )  
 if node['id'] in self.state['expanded']:  
 icon.style.add(transform="rotate(90deg)")  
 icon.add(Element('span', text="▶"))  
 header.add(icon)  
   
 # Node icon (if provided)  
 if 'icon' in node:  
 node\_icon = Element('span')  
 node\_icon.style.add(margin\_right="0.5rem")  
 node\_icon.add(Element('span', text=node['icon']))  
 header.add(node\_icon)  
   
 # Node label  
 label = Element('span')  
 label.add(Element('span', text=node['label']))  
 header.add(label)  
   
 # Add click handler for expansion toggle  
 if has\_children:  
 header.on('click', lambda: self.\_toggle\_item(node['id']))  
   
 node\_container.add(header)  
   
 # Render children if node is expanded  
 if has\_children and node['id'] in self.state['expanded']:  
 children\_container = Element('div')  
 for child in node['children']:  
 children\_container.add(self.\_render\_node(child, level + 1))  
 node\_container.add(children\_container)  
   
 return node\_container  
  
class VirtualList(Component):  
 """虚拟滚动列表组件，用于高效渲染大量数据"""  
   
 def \_\_init\_\_(self,   
 items: List[Any],  
 render\_item: Callable[[Any], Component],  
 item\_height: int = 40,  
 container\_height: int = 400,  
 buffer\_size: int = 5):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('items', items)  
 self.set\_prop('render\_item', render\_item)  
 self.set\_prop('item\_height', item\_height)  
 self.set\_prop('container\_height', container\_height)  
 self.set\_prop('buffer\_size', buffer\_size)  
   
 self.state.update({  
 'scroll\_top': 0,  
 'visible\_items': [],  
 'total\_height': len(items) \* item\_height,  
 'padding\_top': 0,  
 'padding\_bottom': 0  
 })  
   
 self.style.add(  
 height=f"{container\_height}px",  
 overflow\_y="auto",  
 position="relative"  
 )  
   
 self.on\_scroll = EventDelegate()  
 self.on\_scroll.add(self.\_handle\_scroll)  
   
 def \_handle\_scroll(self, event: Dict[str, Any]):  
 """处理滚动事件"""  
 scroll\_top = event['target'].scrollTop  
 self.\_update\_visible\_items(scroll\_top)  
   
 def \_update\_visible\_items(self, scroll\_top: int):  
 """更新可见项目列表"""  
 self.state['scroll\_top'] = scroll\_top  
   
 # 计算可见范围  
 start\_index = max(0, scroll\_top // self.props['item\_height'] - self.props['buffer\_size'])  
 visible\_count = (self.props['container\_height'] // self.props['item\_height'] +   
 2 \* self.props['buffer\_size'])  
 end\_index = min(len(self.props['items']), start\_index + visible\_count)  
   
 # 更新可见项目  
 self.state['visible\_items'] = self.props['items'][start\_index:end\_index]  
   
 # 更新padding以保持滚动位置  
 self.state['padding\_top'] = start\_index \* self.props['item\_height']  
 self.state['padding\_bottom'] = (  
 (len(self.props['items']) - end\_index) \* self.props['item\_height']  
 )  
   
 def render(self):  
 """渲染虚拟列表"""  
 # 容器  
 container = Component()  
 container.tag\_name = "div"  
 container.style.add(  
 height="100%",  
 overflow\_y="auto"  
 )  
   
 # 内容包装器  
 content = Component()  
 content.tag\_name = "div"  
 content.style.add(  
 position="relative",  
 height=f"{self.state['total\_height']}px"  
 )  
   
 # 可见项目容器  
 items\_container = Component()  
 items\_container.tag\_name = "div"  
 items\_container.style.add(  
 position="absolute",  
 top=f"{self.state['padding\_top']}px",  
 left="0",  
 right="0"  
 )  
   
 # 渲染可见项目  
 for item in self.state['visible\_items']:  
 rendered\_item = self.props['render\_item'](item)  
 rendered\_item.style.add(  
 height=f"{self.props['item\_height']}px"  
 )  
 items\_container.add\_child(rendered\_item)  
   
 content.add\_child(items\_container)  
 container.add\_child(content)  
   
 return container  
  
class DraggableList(Component):  
 """可拖放的列表组件"""  
   
 def \_\_init\_\_(self,   
 items: list[Any],  
 render\_item: Optional[Callable[[Any], Component]] = None,  
 on\_reorder: Optional[Callable[[list[Any]], None]] = None):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('items', items)  
 self.set\_prop('render\_item', render\_item or self.\_default\_render\_item)  
 self.set\_prop('on\_reorder', on\_reorder)  
   
 self.state.update({  
 'dragging\_index': None,  
 'drag\_over\_index': None,  
 'items': items.copy()  
 })  
   
 # 设置容器样式  
 self.style.add(  
 position="relative",  
 user\_select="none"  
 )  
   
 def \_default\_render\_item(self, item: Any) -> Component:  
 """默认项渲染器"""  
 text = Text(str(item))  
 text.style.add(  
 padding="1rem",  
 background\_color="#ffffff",  
 border="1px solid #e0e0e0",  
 margin\_bottom="0.5rem",  
 cursor="move"  
 )  
 return text  
   
 def \_handle\_drag\_start(self, index: int, event: dict[str, Any]):  
 """处理拖拽开始事件"""  
 try:  
 self.state['dragging\_index'] = index  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error handling drag start: {e}", exc\_info=True)  
   
 def \_handle\_drag\_over(self, index: int, event: dict[str, Any]):  
 """处理拖拽悬停事件"""  
 try:  
 if index != self.state['drag\_over\_index']:  
 self.state['drag\_over\_index'] = index  
 self.\_update()  
 except Exception as e:  
 self.\_logger.error(f"Error handling drag over: {e}", exc\_info=True)  
   
 def \_handle\_drop(self, index: int, event: dict[str, Any]):  
 """处理放置事件"""  
 try:  
 dragging\_index = self.state['dragging\_index']  
 if dragging\_index is not None and dragging\_index != index:  
 items = self.state['items']  
 item = items.pop(dragging\_index)  
 items.insert(index, item)  
   
 if self.props['on\_reorder']:  
 self.props['on\_reorder'](items)  
   
 self.state.update({  
 'dragging\_index': None,  
 'drag\_over\_index': None  
 })  
 self.\_update()  
   
 except Exception as e:  
 self.\_logger.error(f"Error handling drop: {e}", exc\_info=True)  
   
 def render(self) -> Element:  
 """渲染可拖放列表"""  
 try:  
 container = super().render()  
 items = self.state['items']  
 dragging\_index = self.state['dragging\_index']  
 drag\_over\_index = self.state['drag\_over\_index']  
   
 for i, item in enumerate(items):  
 item\_container = Element('div')  
 item\_container.style.add(  
 opacity="1" if i != dragging\_index else "0.5",  
 transform="none" if i != drag\_over\_index else "translateY(8px)",  
 transition="transform 0.15s ease-in-out"  
 )  
   
 # 添加拖放事件监听器  
 item\_container.set\_attribute('draggable', 'true')  
 item\_container.add\_event\_listener('dragstart', lambda e, i=i: self.\_handle\_drag\_start(i, e))  
 item\_container.add\_event\_listener('dragover', lambda e, i=i: self.\_handle\_drag\_over(i, e))  
 item\_container.add\_event\_listener('drop', lambda e, i=i: self.\_handle\_drop(i, e))  
   
 # 渲染项内容  
 item\_content = self.props['render\_item'](item)  
 item\_container.append\_child(item\_content.render())  
   
 container.append\_child(item\_container)  
   
 return container  
   
 except Exception as e:  
 self.\_logger.error(f"Error rendering draggable list: {e}", exc\_info=True)  
 raise  
  
class Table(Component):  
 """表格组件"""  
   
 def \_\_init\_\_(self, columns: list[dict[str, str]], data: list[dict[str, Any]],  
 sortable: bool = True, filterable: bool = True,  
 page\_size: int = 10):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('columns', columns) # [{"key": "id", "title": "ID"}, ...]  
 self.set\_prop('data', data)  
 self.set\_prop('sortable', sortable)  
 self.set\_prop('filterable', filterable)  
 self.set\_prop('page\_size', page\_size)  
 self.set\_prop('current\_page', 1)  
   
 # State for sorting and filtering  
 self.state['sort\_key'] = None  
 self.state['sort\_order'] = 'asc'  
 self.state['filters'] = {}  
   
 def render(self):  
 container = Element('div')  
   
 # Create table element  
 table = Element('table')  
 table.style.add(  
 width="100%",  
 border\_collapse="collapse",  
 margin="1rem 0"  
 )  
   
 # Render header  
 header = Element('thead')  
 header\_row = Element('tr')  
   
 for col in self.props['columns']:  
 th = Element('th')  
 th.style.add(  
 padding="0.75rem",  
 border\_bottom="2px solid #ddd",  
 text\_align="left",  
 font\_weight="bold"  
 )  
   
 if self.props['sortable']:  
 sort\_container = Element('div')  
 sort\_container.style.add(  
 display="flex",  
 align\_items="center",  
 cursor="pointer"  
 )  
 sort\_container.add(Element('span', text=col['title']))  
 sort\_container.add(Element('span', text="↕️", style={"margin-left": "0.5rem"}))  
 th.add(sort\_container)  
 else:  
 th.add(Element('span', text=col['title']))  
   
 header\_row.add(th)  
   
 header.add(header\_row)  
 table.add(header)  
   
 # Render body  
 body = Element('tbody')  
   
 # Apply pagination  
 start\_idx = (self.props['current\_page'] - 1) \* self.props['page\_size']  
 end\_idx = start\_idx + self.props['page\_size']  
 page\_data = self.props['data'][start\_idx:end\_idx]  
   
 for row\_data in page\_data:  
 tr = Element('tr')  
 tr.style.add(  
 border\_bottom="1px solid #ddd",  
 transition="background-color 0.2s"  
 )  
 tr.add\_hover\_style(background\_color="#f5f5f5")  
   
 for col in self.props['columns']:  
 td = Element('td')  
 td.style.add(padding="0.75rem")  
 td.add(Element('span', text=str(row\_data.get(col['key'], ''))))  
 tr.add(td)  
   
 body.add(tr)  
   
 table.add(body)  
 container.add(table)  
   
 # Add pagination  
 if len(self.props['data']) > self.props['page\_size']:  
 pagination = self.\_render\_pagination()  
 container.add(pagination)  
   
 return container  
   
 def \_render\_pagination(self):  
 total\_pages = (len(self.props['data']) + self.props['page\_size'] - 1) // self.props['page\_size']  
   
 pagination = Element('div')  
 pagination.style.add(  
 display="flex",  
 justify\_content="center",  
 align\_items="center",  
 margin\_top="1rem"  
 )  
   
 # Previous button  
 prev\_btn = Element('button')  
 prev\_btn.add(Element('span', text="Previous"))  
 prev\_btn.style.add(  
 padding="0.5rem 1rem",  
 margin="0 0.25rem",  
 border="1px solid #ddd",  
 border\_radius="4px",  
 cursor="pointer" if self.props['current\_page'] > 1 else "not-allowed",  
 background\_color="#fff"  
 )  
 pagination.add(prev\_btn)  
   
 # Page numbers  
 for page in range(1, total\_pages + 1):  
 page\_btn = Element('button')  
 page\_btn.add(Element('span', text=str(page)))  
 page\_btn.style.add(  
 padding="0.5rem 1rem",  
 margin="0 0.25rem",  
 border="1px solid #ddd",  
 border\_radius="4px",  
 cursor="pointer",  
 background\_color="#fff" if page != self.props['current\_page'] else "#e6e6e6"  
 )  
 pagination.add(page\_btn)  
   
 # Next button  
 next\_btn = Element('button')  
 next\_btn.add(Element('span', text="Next"))  
 next\_btn.style.add(  
 padding="0.5rem 1rem",  
 margin="0 0.25rem",  
 border="1px solid #ddd",  
 border\_radius="4px",  
 cursor="pointer" if self.props['current\_page'] < total\_pages else "not-allowed",  
 background\_color="#fff"  
 )  
 pagination.add(next\_btn)  
   
 return pagination  
  
class Tree(Component):  
 """树形组件"""  
 def \_\_init\_\_(self, data: List[Dict[str, Any]], expanded: bool = False):  
 """  
 初始化树形组件  
 data: 树形数据，每个节点是一个字典，包含'id'、'label'、'children'等键  
 """  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('data', data)  
 self.state['expanded'] = set() # Store expanded node IDs  
   
 # Expand all nodes if expanded is True  
 if expanded:  
 self.\_expand\_all(data)  
   
 def \_expand\_all(self, nodes: List[Dict[str, Any]]) -> None:  
 """递归展开所有节点"""  
 for node in nodes:  
 self.state['expanded'].add(node['id'])  
 if node.get('children'):  
 self.\_expand\_all(node['children'])  
   
 def toggle\_node(self, node\_id: str) -> None:  
 """Toggle node expansion state"""  
 if node\_id in self.state['expanded']:  
 self.state['expanded'].remove(node\_id)  
 else:  
 self.state['expanded'].add(node\_id)  
 self.\_update()  
   
 def \_render\_node(self, node: Dict[str, Any], level: int = 0) -> Element:  
 """Render a single node and its children"""  
 node\_container = Element('div')  
   
 # Node header  
 header = Element('div')  
 header.style.add(  
 display="flex",  
 align\_items="center",  
 padding="0.5rem",  
 padding\_left=f"{level \* 1.5 + 0.5}rem",  
 cursor="pointer",  
 transition="background-color 0.2s"  
 )  
 header.add\_hover\_style(background\_color="#f5f5f5")  
   
 # Expand/collapse icon  
 has\_children = 'children' in node and node['children']  
 if has\_children:  
 icon = Element('span')  
 icon.style.add(  
 margin\_right="0.5rem",  
 transition="transform 0.2s"  
 )  
 if node['id'] in self.state['expanded']:  
 icon.style.add(transform="rotate(90deg)")  
 icon.add(Element('span', text="▶"))  
 header.add(icon)  
   
 # Node icon (if provided)  
 if 'icon' in node:  
 node\_icon = Element('span')  
 node\_icon.style.add(margin\_right="0.5rem")  
 node\_icon.add(Element('span', text=node['icon']))  
 header.add(node\_icon)  
   
 # Node label  
 label = Element('span')  
 label.add(Element('span', text=node['label']))  
 header.add(label)  
   
 # Add click handler for expansion toggle  
 if has\_children:  
 header.on('click', lambda: self.toggle\_node(node['id']))  
   
 node\_container.add(header)  
   
 # Render children if node is expanded  
 if has\_children and node['id'] in self.state['expanded']:  
 children\_container = Element('div')  
 for child in node['children']:  
 children\_container.add(self.\_render\_node(child, level + 1))  
 node\_container.add(children\_container)  
   
 return node\_container  
   
 def render(self):  
 container = Element('div')  
 container.style.add(  
 border="1px solid #ddd",  
 border\_radius="4px",  
 overflow="hidden"  
 )  
   
 # Render each root node  
 for node in self.props['data']:  
 container.add(self.\_render\_node(node))  
   
 return container  
  
class Responsive(Component):  
 """响应式容器组件"""  
 breakpoints = {  
 'sm': '576px',  
 'md': '768px',  
 'lg': '992px',  
 'xl': '1200px',  
 'xxl': '1400px'  
 }  
   
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.style.add(  
 width="100%",  
 margin="0 auto",  
 padding="0 15px",  
 box\_sizing="border-box"  
 )  
   
 def add\_media\_query(self, breakpoint: str, styles: dict[str, str]):  
 self.style.add\_media\_query(  
 f"(min-width: {self.breakpoints[breakpoint]})",  
 styles  
 )  
 return self  
  
class Skeleton(Component):  
 """骨架屏组件"""  
 def \_\_init\_\_(self, type: str = "text", rows: int = 1, height: str = "1rem"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('type', type)  
 self.set\_prop('rows', rows)  
 self.set\_prop('height', height)  
 self.style.add(  
 background="linear-gradient(90deg, #f0f0f0 25%, #e0e0e0 50%, #f0f0f0 75%)",  
 background\_size="200% 100%",  
 animation="skeleton-loading 1.5s infinite",  
 border\_radius="4px",  
 height=height,  
 margin\_bottom="0.5rem"  
 )  
  
class Carousel(Component):  
 """幻灯片组件"""  
 def \_\_init\_\_(self, images: list[dict[str, str]], auto\_play: bool = True, interval: int = 3000):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('images', images) # [{"src": "...", "alt": "..."}]  
 self.set\_prop('auto\_play', auto\_play)  
 self.set\_prop('interval', interval)  
 self.state['current\_index'] = 0  
 self.style.add(  
 position="relative",  
 overflow="hidden",  
 width="100%",  
 height="100%"  
 )  
  
class Drawer(Component):  
 """抽屉组件"""  
 def \_\_init\_\_(self, content: Component, position: str = "left", width: str = "300px"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('content', content)  
 self.set\_prop('position', position)  
 self.set\_prop('width', width)  
 self.state['visible'] = False  
 self.style.add(  
 position="fixed",  
 top="0",  
 height="100%",  
 background\_color="#ffffff",  
 box\_shadow="0 0 10px rgba(0,0,0,0.1)",  
 transition="transform 0.3s ease-in-out",  
 z\_index="1000"  
 )  
  
class Progress(Component):  
 """进度条组件"""  
 def \_\_init\_\_(self, value: int = 0, max: int = 100, type: str = "bar", color: str = "#007bff"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('value', value)  
 self.set\_prop('max', max)  
 self.set\_prop('type', type)  
 self.set\_prop('color', color)  
 self.style.add(  
 width="100%",  
 height="0.5rem",  
 background\_color="#e9ecef",  
 border\_radius="0.25rem",  
 overflow="hidden"  
 )  
  
class Badge(Component):  
 """徽章组件"""  
 def \_\_init\_\_(self, text: str, type: str = "primary", pill: bool = False):  
 super().\_\_init\_\_()  
 self.tag\_name = "span"  
 self.set\_prop('text', text)  
 self.set\_prop('type', type)  
 self.set\_prop('pill', pill)  
 self.style.add(  
 display="inline-block",  
 padding="0.25em 0.4em",  
 font\_size="75%",  
 font\_weight="700",  
 line\_height="1",  
 text\_align="center",  
 white\_space="nowrap",  
 vertical\_align="baseline",  
 border\_radius="0.25rem" if not pill else "10rem",  
 color="#fff",  
 background\_color=self.\_get\_type\_color(type)  
 )  
  
 def \_get\_type\_color(self, type: str) -> str:  
 colors = {  
 'primary': '#007bff',  
 'secondary': '#6c757d',  
 'success': '#28a745',  
 'danger': '#dc3545',  
 'warning': '#ffc107',  
 'info': '#17a2b8'  
 }  
 return colors.get(type, colors['primary'])  
  
class Tooltip(Component):  
 """提示框组件"""  
 def \_\_init\_\_(self, content: str, position: str = "top"):  
 super().\_\_init\_\_()  
 self.tag\_name = "div"  
 self.set\_prop('content', content)  
 self.set\_prop('position', position)  
 self.style.add(  
 position="relative",  
 display="inline-block"  
 )

config.py

import os  
from dotenv import load\_dotenv  
  
basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))  
load\_dotenv()  
  
class Config:  
 SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'dev-key-please-change-in-production'  
 SQLALCHEMY\_DATABASE\_URI = os.environ.get('DATABASE\_URL') or \  
 'sqlite:///' + os.path.join(basedir, 'app.db')  
 SQLALCHEMY\_TRACK\_MODIFICATIONS = False

core.py

"""  
Core functionality for PytoWeb framework  
"""  
  
from typing import Dict, Any, Optional, Type  
from .components import Component  
from .vdom import VNode, VDOMRenderer, VDOMDiffer  
from .events import EventBridge, StateManager  
from .styles import Style  
from .themes import Theme, ThemeProvider  
from .animations import AnimationManager  
  
class PytoWeb:  
 """Main PytoWeb framework class"""  
   
 def \_\_init\_\_(self):  
 self.state\_manager = StateManager()  
 self.event\_bridge = EventBridge()  
 self.theme = Theme()  
 ThemeProvider.set\_theme(self.theme)  
   
 def create\_app(self, root\_component: Type[Component], props: Dict[str, Any] = None) -> str:  
 """Create a new PytoWeb application"""  
 # Initialize root component  
 instance = root\_component()  
 if props:  
 for key, value in props.items():  
 instance.set\_prop(key, value)  
   
 # Generate HTML  
 html = self.\_generate\_html(instance)  
   
 # Add framework scripts  
 html += self.\_get\_framework\_scripts()  
   
 return html  
   
 def \_generate\_html(self, root: Component) -> str:  
 """Generate HTML from component tree"""  
 # Get all registered animations CSS  
 animations\_css = AnimationManager.get\_all\_css()  
   
 # Create HTML template  
 html = f"""  
 <!DOCTYPE html>  
 <html>  
 <head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <style>  
 {animations\_css}  
 {self.\_get\_default\_styles()}  
 </style>  
 </head>  
 <body>  
 <div id="app">  
 {root.render()}  
 </div>  
 </body>  
 </html>  
 """  
 return html  
   
 def \_get\_framework\_scripts(self) -> str:  
 """Get framework JavaScript code"""  
 return f"""  
 <script>  
 {EventBridge.get\_client\_script()}  
   
 // Framework initialization  
 document.addEventListener('DOMContentLoaded', function() {{  
 console.log('PytoWeb initialized');  
 }});  
 </script>  
 """  
   
 def \_get\_default\_styles(self) -> str:  
 """Get default framework styles"""  
 from .styles import DEFAULT\_STYLES  
 return DEFAULT\_STYLES  
   
 def handle\_event(self, event\_type: str, event\_data: Dict[str, Any]):  
 """Handle framework events"""  
 self.event\_bridge.handle\_event(event\_type, event\_data)  
   
 def set\_theme(self, theme: Theme):  
 """Set framework theme"""  
 self.theme = theme  
 ThemeProvider.set\_theme(theme)  
   
 def get\_state\_manager(self) -> StateManager:  
 """Get state manager instance"""  
 return self.state\_manager  
   
 @staticmethod  
 def create\_style(\*\*styles) -> Style:  
 """Create a new style instance"""  
 return Style(\*\*styles)  
   
 @staticmethod  
 def register\_animation(name: str, keyframes: Dict[str, Dict[str, str]]):  
 """Register a new animation"""  
 from .animations import Animation  
 animation = Animation(name, keyframes)  
 AnimationManager.register(animation)

elements.py

"""  
Core module for HTML element creation and manipulation  
"""  
  
from \_\_future\_\_ import annotations  
from typing import List, Dict, Any, Optional, Callable, Union, TypeVar, TYPE\_CHECKING  
from dataclasses import dataclass  
from .styles import Style  
  
T = TypeVar('T', bound='Element')  
  
class ElementError(Exception):  
 """Element creation or manipulation error"""  
 pass  
  
@dataclass  
class EventHandler:  
 """Event handler container"""  
 name: str  
 handler: Callable[..., Any]  
   
 def \_\_post\_init\_\_(self):  
 if not callable(self.handler):  
 raise ElementError("Event handler must be callable")  
  
class Element:  
 """Base class for all HTML elements"""  
   
 VOID\_ELEMENTS = {  
 'area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input',  
 'link', 'meta', 'param', 'source', 'track', 'wbr'  
 }  
   
 def \_\_init\_\_(self, tag: str, text: str = "", \*\*attributes: Any):  
 """Initialize element"""  
 try:  
 if not isinstance(tag, str):  
 raise ElementError("Tag must be a string")  
 if not tag:  
 raise ElementError("Tag cannot be empty")  
   
 self.tag = tag.lower() # Normalize tag name  
 self.text = str(text) # Ensure text is string  
 self.attributes: Dict[str, Any] = {}  
 self.children: List[Element] = []  
 self.style = Style() # Use Style class for style management  
 self.events: Dict[str, EventHandler] = {}  
   
 # Process attributes  
 for key, value in attributes.items():  
 if value is not None: # Only add non-None attributes  
 self.attributes[key] = str(value)  
   
 except Exception as e:  
 raise ElementError(f"Failed to initialize element: {e}") from e  
   
 def add(self, child: Union[Element, str]) -> T:  
 """Add child element"""  
 try:  
 if isinstance(child, str):  
 child = Element('span', text=child)  
 elif not isinstance(child, Element):  
 raise ElementError("Child must be an Element or string")  
   
 self.children.append(child)  
 return self  
   
 except Exception as e:  
 raise ElementError(f"Failed to add child: {e}") from e  
   
 def add\_child(self, child: Union[Element, str]) -> T:  
 """Alias for add()"""  
 return self.add(child)  
   
 def to\_html(self) -> str:  
 """Convert element to HTML string"""  
 try:  
 print(f"[DEBUG] Converting {self.tag} to HTML")  
 # Start tag  
 html = [f"<{self.tag}"]  
   
 # Add attributes  
 for key, value in self.attributes.items():  
 html.append(f' {key}="{value}"')  
   
 # Add style  
 if self.style and self.style.get\_all():  
 style\_str = '; '.join(f"{k}: {v}" for k, v in self.style.get\_all().items())  
 html.append(f' style="{style\_str}"')  
   
 # Close start tag  
 html.append('>')  
   
 # Add text content  
 if self.text:  
 html.append(self.text)  
   
 # Add children  
 for child in self.children:  
 try:  
 child\_html = child.to\_html()  
 html.append(child\_html)  
 except Exception as e:  
 print(f"[ERROR] Failed to render child of {self.tag}: {e}")  
 raise  
   
 # Add closing tag if not void element  
 if self.tag not in self.VOID\_ELEMENTS:  
 html.append(f"</{self.tag}>")  
   
 result = ''.join(html)  
 print(f"[DEBUG] Generated HTML for {self.tag}: {result[:100]}...")  
 return result  
   
 except Exception as e:  
 print(f"[ERROR] Failed to generate HTML for {self.tag}: {e}")  
 raise ElementError(f"Failed to generate HTML: {e}") from e  
   
 def \_\_str\_\_(self) -> str:  
 """String representation is HTML"""  
 return self.to\_html()  
  
# Convenience functions for creating common elements  
def div(\*children: Element, \*\*attrs: Any) -> Element:  
 """Create a div element"""  
 return Element('div', \*\*attrs).add(\*children)  
   
def span(\*children: Element, \*\*attrs: Any) -> Element:  
 """Create a span element"""  
 return Element('span', \*\*attrs).add(\*children)  
   
def p(\*children: Element, \*\*attrs: Any) -> Element:  
 """Create a paragraph element"""  
 return Element('p', \*\*attrs).add(\*children)  
   
def a(href: str, \*children: Element, \*\*attrs: Any) -> Element:  
 """Create an anchor element"""  
 attrs['href'] = href  
 return Element('a', \*\*attrs).add(\*children)  
   
def img(src: str, alt: str = "", \*\*attrs: Any) -> Element:  
 """Create an image element"""  
 attrs.update({'src': src, 'alt': alt})  
 return Element('img', \*\*attrs)  
   
def button(\*children: Element, \*\*attrs: Any) -> Element:  
 """Create a button element"""  
 return Element('button', \*\*attrs).add(\*children)  
   
def input(type: str = "text", \*\*attrs: Any) -> Element:  
 """Create an input element"""  
 attrs['type'] = type  
 return Element('input', \*\*attrs)

events.py

from \_\_future\_\_ import annotations  
from .state import StateManager  
  
"""  
PytoWeb事件系统  
  
提供事件处理、委托、批处理和状态管理功能。  
"""  
  
from typing import (  
 Dict, Any, Optional, Callable, List, Set,  
 TypeVar, TypedDict, Union  
)  
from collections import defaultdict  
import asyncio  
import time  
import uuid  
import json  
import weakref  
import logging  
from concurrent.futures import ThreadPoolExecutor  
  
# 配置日志  
logging.basicConfig(level=logging.DEBUG)  
logger = logging.getLogger(\_\_name\_\_)  
  
# 类型别名  
T = TypeVar('T')  
EventType = str  
HandlerType = Callable[..., None]  
EventData = Dict[str, Any]  
  
class Event:  
 """事件基类"""  
 def \_\_init\_\_(self, event\_type: EventType, target: Any, data: EventData | None = None):  
 self.type = event\_type  
 self.target = target  
 self.data = data or {}  
 self.timestamp = time.time()  
 self.propagation\_stopped = False  
 self.default\_prevented = False  
   
 def stop\_propagation(self):  
 """停止事件传播"""  
 self.propagation\_stopped = True  
   
 def prevent\_default(self):  
 """阻止默认行为"""  
 self.default\_prevented = True  
  
class EventHandler:  
 """事件处理器"""  
 def \_\_init\_\_(self,   
 callback: Callable[[Event], None],  
 once: bool = False,  
 capture: bool = False,  
 passive: bool = False):  
 self.callback = callback  
 self.once = once  
 self.capture = capture  
 self.passive = passive  
  
class EventBridge:  
 """Python和JavaScript事件桥接器"""  
   
 \_handlers: Dict[str, Callable] = {}  
 \_js\_code = """  
 window.pytoweb = {  
 handlers: {},  
   
 handleEvent: function(handlerId, event) {  
 // 发送事件数据到Python  
 const eventData = {  
 type: event.type,  
 target: {  
 id: event.target.id,  
 value: event.target.value,  
 checked: event.target.checked,  
 dataset: event.target.dataset,  
 scrollTop: event.target.scrollTop,  
 scrollHeight: event.target.scrollHeight,  
 clientHeight: event.target.clientHeight  
 },  
 clientX: event.clientX,  
 clientY: event.clientY,  
 timestamp: Date.now()  
 };  
   
 // 发送到Python后端  
 this.sendToPython(handlerId, eventData);  
 },  
   
 sendToPython: async function(handlerId, data) {  
 try {  
 const response = await fetch('/api/events', {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 body: JSON.stringify({  
 handlerId: handlerId,  
 data: data  
 })  
 });  
   
 if (!response.ok) {  
 throw new Error('Event handling failed');  
 }  
 } catch (error) {  
 console.error('Error sending event to Python:', error);  
 }  
 },  
   
 registerHandler: function(handlerId, options = {}) {  
 const handler = (event) => {  
 if (options.preventDefault) {  
 event.preventDefault();  
 }  
 if (options.stopPropagation) {  
 event.stopPropagation();  
 }  
 this.handleEvent(handlerId, event);  
 };  
   
 this.handlers[handlerId] = handler;  
 return handler;  
 },  
   
 removeHandler: function(handlerId) {  
 delete this.handlers[handlerId];  
 }  
 };  
 """  
   
 @classmethod  
 def register\_handler(cls, handler: Callable) -> str:  
 """注册事件处理器并返回处理器ID"""  
 handler\_id = str(uuid.uuid4())  
 cls.\_handlers[handler\_id] = handler  
 return handler\_id  
   
 @classmethod  
 def remove\_handler(cls, handler\_id: str):  
 """移除事件处理器"""  
 if handler\_id in cls.\_handlers:  
 del cls.\_handlers[handler\_id]  
   
 @classmethod  
 def handle\_event(cls, handler\_id: str, event\_data: Dict[str, Any]):  
 """处理从JavaScript发来的事件"""  
 if handler\_id in cls.\_handlers:  
 try:  
 event = Event(  
 event\_type=event\_data.get('type', ''),  
 target=event\_data.get('target', {}),  
 data=event\_data  
 )  
 cls.\_handlers[handler\_id](event)  
 except Exception as e:  
 logger.error(f"Error handling event: {e}", exc\_info=True)  
  
class EventDelegate:  
 """事件委托类"""  
 def \_\_init\_\_(self):  
 self.\_handlers: list[HandlerType] = []  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
   
 def add(self, handler: HandlerType):  
 """添加事件处理器"""  
 if handler not in self.\_handlers:  
 self.\_handlers.append(handler)  
   
 def remove(self, handler: HandlerType):  
 """移除事件处理器"""  
 if handler in self.\_handlers:  
 self.\_handlers.remove(handler)  
   
 def clear(self):  
 """清除所有处理器"""  
 self.\_handlers.clear()  
   
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 """调用所有处理器"""  
 for handler in self.\_handlers:  
 try:  
 handler(\*args, \*\*kwargs)  
 except Exception as e:  
 self.\_logger.error(f"Error in event handler: {e}", exc\_info=True)  
  
class EventEmitter:  
 """增强的事件发射器，支持事件委托和批处理"""  
   
 def \_\_init\_\_(self):  
 self.\_handlers: dict[str, list[EventHandler]] = defaultdict(list)  
 self.\_delegate\_handlers: dict[str, dict[str, list[EventHandler]]] = defaultdict(lambda: defaultdict(list))  
 self.\_batch\_handlers: dict[str, list[Callable[[list[Event]], None]]] = defaultdict(list)  
 self.\_batch\_queue: dict[str, list[Event]] = defaultdict(list)  
 self.\_batch\_timeout = 16.67 # ~60fps  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
   
 def on(self, event\_type: str, callback: Callable[[Event], None], selector: str | None = None, \*\*options):  
 """添加事件监听器"""  
 handler = EventHandler(callback, \*\*options)  
   
 if selector:  
 self.\_delegate\_handlers[event\_type][selector].append(handler)  
 else:  
 self.\_handlers[event\_type].append(handler)  
   
 def off(self, event\_type: str, callback: Callable[[Event], None] | None = None, selector: str | None = None):  
 """移除事件监听器"""  
 if selector:  
 if callback:  
 self.\_delegate\_handlers[event\_type][selector] = [  
 h for h in self.\_delegate\_handlers[event\_type][selector]  
 if h.callback != callback  
 ]  
 else:  
 self.\_delegate\_handlers[event\_type][selector].clear()  
 else:  
 if callback:  
 self.\_handlers[event\_type] = [  
 h for h in self.\_handlers[event\_type]  
 if h.callback != callback  
 ]  
 else:  
 self.\_handlers[event\_type].clear()  
   
 def emit(self, event: Event):  
 """发射事件"""  
 if event.propagation\_stopped:  
 return  
   
 # 处理直接监听器  
 for handler in self.\_handlers[event.type]:  
 try:  
 if handler.once:  
 self.off(event.type, handler.callback)  
 handler.callback(event)  
 except Exception as e:  
 self.\_logger.error(f"Error in event handler: {e}", exc\_info=True)  
   
 # 处理委托监听器  
 if isinstance(event.target, dict) and 'id' in event.target:  
 target\_id = event.target['id']  
 for selector, handlers in self.\_delegate\_handlers[event.type].items():  
 if self.\_matches\_selector(target\_id, selector):  
 for handler in handlers:  
 try:  
 if handler.once:  
 self.off(event.type, handler.callback, selector)  
 handler.callback(event)  
 except Exception as e:  
 self.\_logger.error(f"Error in delegate handler: {e}", exc\_info=True)  
   
 # 添加到批处理队列  
 if event.type in self.\_batch\_handlers:  
 self.\_batch\_queue[event.type].append(event)  
 self.\_schedule\_batch\_process(event.type)  
   
 def \_matches\_selector(self, target\_id: str, selector: str) -> bool:  
 """检查目标是否匹配选择器"""  
 # 简单的选择器匹配实现  
 return selector.startswith('#') and target\_id == selector[1:]  
   
 def add\_batch\_handler(self, event\_type: str, handler: Callable[[list[Event]], None]):  
 """添加批处理事件处理器"""  
 self.\_batch\_handlers[event\_type].append(handler)  
   
 def \_schedule\_batch\_process(self, event\_type: str):  
 """调度批处理"""  
 async def process\_batch():  
 await asyncio.sleep(self.\_batch\_timeout / 1000)  
 events = self.\_batch\_queue[event\_type]  
 self.\_batch\_queue[event\_type] = []  
   
 for handler in self.\_batch\_handlers[event\_type]:  
 try:  
 handler(events)  
 except Exception as e:  
 self.\_logger.error(f"Error in batch handler: {e}", exc\_info=True)  
   
 asyncio.create\_task(process\_batch())  
  
class EventManager:  
 """全局事件管理系统"""  
   
 \_instance = None  
   
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
   
 def \_\_init\_\_(self):  
 if not hasattr(self, 'initialized'):  
 self.emitter = EventEmitter()  
 self.\_listeners: Dict[str, Set[weakref.ref]] = defaultdict(set)  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
 self.initialized = True  
   
 def add\_listener(self, target: Any, event\_type: str):  
 """添加全局事件监听器"""  
 ref = weakref.ref(target, lambda r: self.\_cleanup\_listener(r, event\_type))  
 self.\_listeners[event\_type].add(ref)  
   
 def remove\_listener(self, target: Any, event\_type: str):  
 """移除全局事件监听器"""  
 to\_remove = None  
 for ref in self.\_listeners[event\_type]:  
 if ref() is target:  
 to\_remove = ref  
 break  
 if to\_remove:  
 self.\_listeners[event\_type].remove(to\_remove)  
   
 def \_cleanup\_listener(self, ref: weakref.ref, event\_type: str):  
 """清理失效的监听器"""  
 self.\_listeners[event\_type].discard(ref)  
   
 def dispatch\_event(self, event: Event, batch: bool = False):  
 """分发事件到所有监听器"""  
 try:  
 if batch:  
 self.emitter.add\_batch\_handler(event.type, lambda events: self.\_dispatch\_batch(events))  
 self.emitter.emit(event)  
 else:  
 self.\_dispatch\_single(event)  
 except Exception as e:  
 self.\_logger.error(f"Error dispatching event: {e}", exc\_info=True)  
   
 def \_dispatch\_single(self, event: Event):  
 """分发单个事件"""  
 for ref in list(self.\_listeners[event.type]):  
 target = ref()  
 if target is not None:  
 try:  
 target.handle\_event(event)  
 except Exception as e:  
 self.\_logger.error(f"Error in event handler: {e}", exc\_info=True)  
   
 def \_dispatch\_batch(self, events: List[Event]):  
 """分发事件批次"""  
 if not events:  
 return  
   
 event\_type = events[0].type  
 for ref in list(self.\_listeners[event\_type]):  
 target = ref()  
 if target is not None:  
 try:  
 target.handle\_event\_batch(events)  
 except Exception as e:  
 self.\_logger.error(f"Error in batch event handler: {e}", exc\_info=True)  
   
 def dispatch\_batch(self, events: List[Event]):  
 """一次性分发多个事件"""  
 for event in events:  
 self.dispatch\_event(event, batch=True)

layouts.py

"""  
Layout system for PytoWeb  
"""  
  
from typing import List, Optional, Union, Tuple, Dict  
from .components import Component, Container  
from .elements import Element  
from .styles import Style  
  
class Grid(Container):  
 """Grid layout component"""  
   
 def \_\_init\_\_(self, columns: int = 12, gap: str = '1rem'):  
 super().\_\_init\_\_()  
 self.columns = columns  
 self.gap = gap  
   
 def render(self) -> Element:  
 container = Element('div')  
 container.style(  
 display='grid',  
 grid\_template\_columns=f'repeat({self.columns}, 1fr)',  
 gap=self.gap  
 )  
   
 for child in self.children:  
 container.add(child.render())  
   
 return container  
  
class Flex(Container):  
 """Flexbox layout component"""  
   
 def \_\_init\_\_(self, direction: str = 'row', justify: str = 'flex-start',   
 align: str = 'stretch', wrap: str = 'nowrap'):  
 super().\_\_init\_\_()  
 self.direction = direction  
 self.justify = justify  
 self.align = align  
 self.wrap = wrap  
   
 def render(self) -> Element:  
 container = Element('div')  
 container.style(  
 display='flex',  
 flex\_direction=self.direction,  
 justify\_content=self.justify,  
 align\_items=self.align,  
 flex\_wrap=self.wrap  
 )  
   
 for child in self.children:  
 container.add(child.render())  
   
 return container  
  
class Responsive(Container):  
 """Responsive layout component"""  
   
 def \_\_init\_\_(self, breakpoints: Dict[str, str] = None):  
 super().\_\_init\_\_()  
 self.breakpoints = breakpoints or {  
 'sm': '576px',  
 'md': '768px',  
 'lg': '992px',  
 'xl': '1200px'  
 }  
   
 def render(self) -> Element:  
 container = Element('div')  
 container.style(  
 width='100%',  
 margin='0 auto',  
 padding='0 15px'  
 )  
   
 # Add media queries for responsive behavior  
 for size, width in self.breakpoints.items():  
 container.style(\*\*{  
 f'@media (min-width: {width})': {  
 'max-width': width  
 }  
 })  
   
 for child in self.children:  
 container.add(child.render())  
   
 return container

router.py

"""  
PytoWeb路由模块  
"""  
  
from \_\_future\_\_ import annotations  
from typing import Dict, List, Callable, Any, Optional, Union, TypeVar, TYPE\_CHECKING  
from dataclasses import dataclass  
import re  
import logging  
from http import HTTPStatus  
  
class RouterError(Exception):  
 """路由错误"""  
 pass  
  
@dataclass  
class Route:  
 """路由定义类"""  
 path: str  
 handler: Callable[..., Any]  
 methods: List[str]  
 name: Optional[str] = None  
  
 def \_\_post\_init\_\_(self):  
 """验证路由参数"""  
 if not self.path.startswith('/'):  
 raise RouterError(f"Path must start with '/': {self.path}")  
 if not callable(self.handler):  
 raise RouterError(f"Handler must be callable: {self.handler}")  
 if not self.methods:  
 self.methods = ['GET']  
 self.methods = [m.upper() for m in self.methods]  
 for method in self.methods:  
 if method not in ['GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'HEAD', 'OPTIONS']:  
 raise RouterError(f"Invalid HTTP method: {method}")  
  
class Router:  
 """路由管理器"""  
   
 def \_\_init\_\_(self):  
 self.routes: List[Route] = []  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
   
 def add(self, path: str, handler: Callable[..., Any], methods: Optional[List[str]] = None, name: Optional[str] = None) -> Router:  
 """添加路由"""  
 try:  
 route = Route(path, handler, methods or ['GET'], name)  
 self.routes.append(route)  
 return self  
 except Exception as e:  
 raise RouterError(f"Failed to add route: {e}") from e  
   
 def route(self, path: str, methods: Union[List[str], str] = 'GET', name: Optional[str] = None) -> Callable:  
 """通用路由装饰器"""  
 if isinstance(methods, str):  
 methods = [methods]  
   
 def decorator(handler: Callable[..., Any]) -> Callable[..., Any]:  
 self.add(path, handler, methods, name)  
 return handler  
 return decorator  
   
 def get(self, path: str, name: Optional[str] = None) -> Callable:  
 """装饰器：添加GET路由"""  
 return self.route(path, ['GET'], name)  
   
 def post(self, path: str, name: Optional[str] = None) -> Callable:  
 """装饰器：添加POST路由"""  
 return self.route(path, ['POST'], name)  
   
 def put(self, path: str, name: Optional[str] = None) -> Callable:  
 """装饰器：添加PUT路由"""  
 return self.route(path, ['PUT'], name)  
   
 def delete(self, path: str, name: Optional[str] = None) -> Callable:  
 """装饰器：添加DELETE路由"""  
 return self.route(path, ['DELETE'], name)  
   
 def match(self, path: str, method: str = 'GET') -> Optional[Route]:  
 """匹配路由"""  
 method = method.upper()  
 try:  
 for route in self.routes:  
 if method in route.methods:  
 # 简单路径匹配  
 if route.path == path:  
 return route  
   
 # 参数路径匹配  
 pattern = re.sub(r'{\w+}', r'([^/]+)', route.path)  
 match = re.match(f'^{pattern}$', path)  
 if match:  
 return route  
   
 return None  
 except Exception as e:  
 self.\_logger.error(f"Route matching error: {e}")  
 return None  
   
 def url\_for(self, name: str, \*\*params: Any) -> str:  
 """根据路由名称生成URL"""  
 try:  
 for route in self.routes:  
 if route.name == name:  
 url = route.path  
 # 替换URL参数  
 for key, value in params.items():  
 placeholder = '{' + key + '}'  
 if placeholder not in url:  
 raise RouterError(f"Parameter '{key}' not found in route '{name}'")  
 url = url.replace(placeholder, str(value))  
 return url  
   
 raise RouterError(f"No route found with name '{name}'")  
 except Exception as e:  
 if isinstance(e, RouterError):  
 raise  
 raise RouterError(f"Failed to generate URL for route '{name}': {e}") from e  
   
 def group(self, prefix: str) -> Router:  
 """创建路由组"""  
 if not prefix.startswith('/'):  
 raise RouterError("Group prefix must start with '/'")  
   
 group\_router = Router()  
   
 def add\_group\_route(path: str, handler: Callable[..., Any], methods: List[str], name: Optional[str] = None) -> None:  
 full\_path = prefix + path  
 self.add(full\_path, handler, methods, name)  
   
 group\_router.add = add\_group\_route  
 return group\_router  
   
 def mount(self, prefix: str, router: Router) -> Router:  
 """挂载其他路由器"""  
 if not prefix.startswith('/'):  
 raise RouterError("Mount prefix must start with '/'")  
   
 try:  
 for route in router.routes:  
 full\_path = prefix + route.path  
 self.add(full\_path, route.handler, route.methods, route.name)  
 return self  
 except Exception as e:  
 raise RouterError(f"Failed to mount router at '{prefix}': {e}") from e  
   
 def middleware(self, middleware\_func: Callable[..., Any]) -> Router:  
 """添加路由中间件"""  
 original\_routes = self.routes[:]  
 for route in original\_routes:  
 original\_handler = route.handler  
   
 def wrapped\_handler(\*args, \*\*kwargs):  
 return middleware\_func(original\_handler, \*args, \*\*kwargs)  
   
 route.handler = wrapped\_handler  
 return self  
  
 def dispatch(self, request):  
 """Dispatch the request to the appropriate handler"""  
 path = request.path  
 method = request.method  
   
 # 查找匹配的路由  
 handler = self.match(path, method)  
 if handler:  
 return handler.handler(request)  
   
 # 没有找到路由  
 return None

run.py

from app import create\_app, db  
  
app = create\_app()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with app.app\_context():  
 db.create\_all()  
 app.run(debug=True)

server.py

"""  
Server module for PytoWeb  
"""  
  
from \_\_future\_\_ import annotations  
from typing import Dict, Any, Optional, Callable, List  
import logging  
import traceback  
from http.server import HTTPServer, BaseHTTPRequestHandler  
import os  
import mimetypes  
import json  
from urllib.parse import parse\_qs, urlparse  
import http  
  
class ServerError(Exception):  
 """Server error"""  
 pass  
  
class RequestHandler(BaseHTTPRequestHandler):  
 """HTTP request handler"""  
   
 def do\_GET(self) -> None:  
 """Handle GET request"""  
 try:  
 # Parse URL  
 parsed\_url = urlparse(self.path)  
 path = parsed\_url.path  
 query = parse\_qs(parsed\_url.query)  
   
 print(f"[DEBUG] Handling GET request for path: {path}")  
 print(f"[DEBUG] Available routes: {list(self.server.routes.keys())}")  
   
 # Find route handler  
 handler = self.server.routes.get(path)  
 if handler:  
 try:  
 print(f"[DEBUG] Found handler for path: {path}")  
 # Call route handler  
 response = handler({"query": query, "method": "GET"})  
 if response:  
 print(f"[DEBUG] Handler returned response: {response[:200]}...")  
 self.send\_response(http.HTTPStatus.OK)  
 self.send\_header('Content-type', 'text/html; charset=utf-8')  
 self.end\_headers()  
 self.wfile.write(response.encode('utf-8'))  
 else:  
 print("[DEBUG] Handler returned None")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR, "Handler returned None")  
 except Exception as e:  
 print(f"[DEBUG] Error in route handler: {e}")  
 self.\_log\_error(f"Error in route handler: {e}")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR)  
 else:  
 print(f"[DEBUG] No handler found for path: {path}")  
 # Try to serve static file  
 static\_file = os.path.join(self.server.static\_dir, path.lstrip('/'))  
 print(f"[DEBUG] Looking for static file: {static\_file}")  
 if os.path.exists(static\_file) and os.path.isfile(static\_file):  
 print(f"[DEBUG] Found static file: {static\_file}")  
 self.serve\_static\_file(static\_file)  
 else:  
 print(f"[DEBUG] Static file not found: {static\_file}")  
 # Return 404  
 self.send\_error(http.HTTPStatus.NOT\_FOUND)  
 except Exception as e:  
 print(f"[DEBUG] Error handling GET request: {e}")  
 self.\_log\_error(f"Error handling GET request: {e}")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR)  
   
 def do\_POST(self) -> None:  
 """Handle POST request"""  
 try:  
 # Read request body  
 content\_length = int(self.headers.get('Content-Length', 0))  
 post\_data = self.rfile.read(content\_length).decode('utf-8')  
   
 try:  
 data = json.loads(post\_data) if post\_data else {}  
 except json.JSONDecodeError:  
 data = parse\_qs(post\_data)  
   
 # Find route handler  
 handler = self.server.routes.get(self.path)  
 if handler:  
 try:  
 # Call route handler  
 response = handler({"data": data, "method": "POST"})  
 if response:  
 self.send\_response(http.HTTPStatus.OK)  
 self.send\_header('Content-type', 'text/html; charset=utf-8')  
 self.end\_headers()  
 self.wfile.write(response.encode('utf-8'))  
 else:  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR, "Handler returned None")  
 except Exception as e:  
 self.\_log\_error(f"Error in route handler: {e}")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR)  
 else:  
 self.send\_error(http.HTTPStatus.NOT\_FOUND)  
 except Exception as e:  
 self.\_log\_error(f"Error handling POST request: {e}")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR)  
   
 def serve\_static\_file(self, filepath: str) -> None:  
 """Serve static file"""  
 try:  
 # Get file MIME type  
 content\_type = self.guess\_type(filepath)  
   
 # Read file content  
 with open(filepath, 'rb') as f:  
 content = f.read()  
   
 # Send response  
 self.send\_response(http.HTTPStatus.OK)  
 self.send\_header('Content-type', content\_type)  
 self.send\_header('Content-length', str(len(content)))  
 self.end\_headers()  
 self.wfile.write(content)  
 except Exception as e:  
 self.\_log\_error(f"Error serving static file: {e}")  
 self.send\_error(http.HTTPStatus.INTERNAL\_SERVER\_ERROR)  
   
 def guess\_type(self, filepath: str) -> str:  
 """Guess file MIME type"""  
 content\_type, \_ = mimetypes.guess\_type(filepath)  
 return content\_type or 'application/octet-stream'  
   
 def \_log\_error(self, message: str) -> None:  
 """Log error"""  
 logging.error(f"{message}\n{traceback.format\_exc()}")  
  
class Server(HTTPServer):  
 """PytoWeb server class"""  
   
 def \_\_init\_\_(self, host: str = "localhost", port: int = 8000):  
 """Initialize server"""  
 super().\_\_init\_\_((host, port), RequestHandler)  
 self.routes: Dict[str, Callable] = {}  
 self.middleware: List[Callable] = []  
 self.static\_dir = os.path.join(os.getcwd(), "static")  
 self.\_logger = logging.getLogger(\_\_name\_\_)  
   
 # Create static file directory  
 if not os.path.exists(self.static\_dir):  
 os.makedirs(self.static\_dir)  
   
 def add\_route(self, path: str, handler: Callable[..., Any]) -> None:  
 """Add route handler"""  
 print(f"[DEBUG] Adding route: {path}")  
 if not path.startswith('/'):  
 path = '/' + path  
 self.routes[path] = handler  
 print(f"[DEBUG] Current routes: {list(self.routes.keys())}")  
   
 def use(self, middleware: Callable[..., Any]) -> None:  
 """Add middleware"""  
 self.middleware.append(middleware)  
   
 def run(self, host: str, port: int) -> None:  
 """Run HTTP server"""  
 try:  
 print(f"[DEBUG] Starting server at http://{host}:{port}")  
 print(f"[DEBUG] Available routes: {list(self.routes.keys())}")  
 self.serve\_forever()  
 except Exception as e:  
 raise ServerError(f"Failed to start server: {e}") from e  
  
 def handle\_error(self, request: Any, client\_address: Any) -> None:  
 """Handle request error"""  
 self.\_logger.error(f"Error handling request from {client\_address}:\n{traceback.format\_exc()}")

state.py

"""State management system for PytoWeb."""  
from typing import Dict, List, Any, Callable  
import json  
import threading  
import time  
from dataclasses import dataclass  
from collections import defaultdict  
  
@dataclass  
class StateChange:  
 """Represents a state change event."""  
 path: str  
 old\_value: Any  
 new\_value: Any  
  
class Store:  
 """Central state store with reactive updates."""  
   
 def \_\_init\_\_(self):  
 self.\_state = {}  
 self.\_subscribers = defaultdict(list)  
 self.\_lock = threading.Lock()  
   
 def get(self, path: str, default: Any = None) -> Any:  
 """Get state value at path."""  
 try:  
 keys = path.split('.')  
 value = self.\_state  
 for key in keys:  
 value = value[key]  
 return value  
 except (KeyError, TypeError):  
 return default  
   
 def set(self, path: str, value: Any):  
 """Set state value at path."""  
 with self.\_lock:  
 keys = path.split('.')  
 target = self.\_state  
   
 # Navigate to the parent of the target  
 for key in keys[:-1]:  
 if key not in target:  
 target[key] = {}  
 target = target[key]  
   
 # Get old value and set new value  
 old\_value = target.get(keys[-1])  
 target[keys[-1]] = value  
   
 # Notify subscribers  
 change = StateChange(path, old\_value, value)  
 self.\_notify\_subscribers(change)  
   
 def subscribe(self, path: str, callback: Callable[[StateChange], None]):  
 """Subscribe to state changes at path."""  
 with self.\_lock:  
 self.\_subscribers[path].append(callback)  
   
 def unsubscribe(self, path: str, callback: Callable[[StateChange], None]):  
 """Unsubscribe from state changes at path."""  
 with self.\_lock:  
 if path in self.\_subscribers:  
 self.\_subscribers[path].remove(callback)  
   
 def \_notify\_subscribers(self, change: StateChange):  
 """Notify all relevant subscribers of state change."""  
 # Notify exact path subscribers  
 for callback in self.\_subscribers[change.path]:  
 callback(change)  
   
 # Notify parent path subscribers  
 parts = change.path.split('.')  
 for i in range(len(parts)):  
 parent\_path = '.'.join(parts[:i])  
 if parent\_path:  
 for callback in self.\_subscribers[parent\_path]:  
 callback(change)  
  
class PersistentStore(Store):  
 """Store with persistence capabilities."""  
   
 def \_\_init\_\_(self, storage\_path: str):  
 super().\_\_init\_\_()  
 self.storage\_path = storage\_path  
 self.\_load\_state()  
   
 def \_load\_state(self):  
 """Load state from storage."""  
 try:  
 with open(self.storage\_path, 'r') as f:  
 self.\_state = json.load(f)  
 except (FileNotFoundError, json.JSONDecodeError):  
 self.\_state = {}  
   
 def \_save\_state(self):  
 """Save state to storage."""  
 with open(self.storage\_path, 'w') as f:  
 json.dump(self.\_state, f)  
   
 def set(self, path: str, value: Any):  
 """Set state value and persist to storage."""  
 super().set(path, value)  
 self.\_save\_state()  
  
class StateManager:  
 """Manages application state and provides reactive updates."""  
   
 \_instance = None  
   
 def \_\_new\_\_(cls):  
 if cls.\_instance is None:  
 cls.\_instance = super().\_\_new\_\_(cls)  
 return cls.\_instance  
   
 def \_\_init\_\_(self, ttl=3600): # 默认1小时过期  
 if not hasattr(self, 'initialized'):  
 self.store = Store()  
 self.\_state = {}  
 self.\_listeners = {}  
 self.\_batch\_updates = False  
 self.\_pending\_updates = {}  
 self.\_ttl = ttl  
 self.\_timestamps = {}  
 self.initialized = True  
   
 def create\_persistent\_store(self, name: str, storage\_path: str) -> PersistentStore:  
 """Create a new persistent store."""  
 store = PersistentStore(storage\_path)  
 setattr(self, f"{name}\_store", store)  
 return store  
   
 def watch(self, paths: List[str], callback: Callable[[StateChange], None]):  
 """Watch multiple paths for changes."""  
 for path in paths:  
 self.store.subscribe(path, callback)  
   
 def unwatch(self, paths: List[str], callback: Callable[[StateChange], None]):  
 """Unwatch multiple paths."""  
 for path in paths:  
 self.store.unsubscribe(path, callback)  
   
 def set(self, key: str, value: Any):  
 if self.\_batch\_updates:  
 self.\_pending\_updates[key] = value  
 return  
   
 if key not in self.\_state or self.\_state[key] != value:  
 self.\_state[key] = value  
 self.\_timestamps[key] = time.time()  
 self.\_notify\_listeners(key)  
   
 def get(self, key: str) -> Any:  
 if key in self.\_state:  
 if time.time() - self.\_timestamps.get(key, 0) > self.\_ttl:  
 self.\_cleanup\_key(key)  
 return None  
 return self.\_state[key]  
 return None  
   
 def \_cleanup\_key(self, key: str):  
 self.\_state.pop(key, None)  
 self.\_timestamps.pop(key, None)  
 if key in self.\_listeners:  
 del self.\_listeners[key]  
   
 def cleanup\_expired(self):  
 current\_time = time.time()  
 expired\_keys = [  
 key for key, timestamp in self.\_timestamps.items()  
 if current\_time - timestamp > self.\_ttl  
 ]  
 for key in expired\_keys:  
 self.\_cleanup\_key(key)  
   
 @classmethod  
 def get\_instance(cls):  
 """Get singleton instance."""  
 return cls()

styles.py

"""  
Modern styling system with advanced features  
"""  
  
from \_\_future\_\_ import annotations  
from typing import Dict, Any, List, Optional, Union, TypeVar, TYPE\_CHECKING  
from dataclasses import dataclass  
  
T = TypeVar('T', bound='Style')  
  
@dataclass  
class StyleUnit:  
 """CSS unit value"""  
 value: Union[int, float]  
 unit: str = 'px'  
   
 def \_\_str\_\_(self) -> str:  
 return f"{self.value}{self.unit}"  
  
class StyleError(Exception):  
 """Style system error"""  
 pass  
  
class Style:  
 """CSS style management class"""  
   
 def \_\_init\_\_(self, \*\*styles: Any):  
 self.rules: Dict[str, str] = {}  
 self.add(\*\*styles)  
   
 def add(self: T, \*\*styles: Any) -> T:  
 """Add CSS styles"""  
 try:  
 for key, value in styles.items():  
 # Convert Python style names to CSS (e.g., font\_size -> font-size)  
 css\_key = key.replace('\_', '-')  
   
 # Handle StyleUnit objects  
 if isinstance(value, StyleUnit):  
 value = str(value)  
 # Handle color tuples (RGB or RGBA)  
 elif isinstance(value, tuple):  
 if len(value) == 3:  
 value = f"rgb({value[0]}, {value[1]}, {value[2]})"  
 elif len(value) == 4:  
 value = f"rgba({value[0]}, {value[1]}, {value[2]}, {value[3]})"  
 else:  
 raise StyleError(f"Invalid color tuple length: {len(value)}")  
 # Handle lists (e.g., for multiple background images)  
 elif isinstance(value, list):  
 value = ', '.join(str(v) for v in value)  
   
 self.rules[css\_key] = str(value)  
 return self  
 except Exception as e:  
 raise StyleError(f"Failed to add styles: {e}") from e  
   
 def remove(self: T, \*keys: str) -> T:  
 """Remove CSS styles"""  
 for key in keys:  
 css\_key = key.replace('\_', '-')  
 self.rules.pop(css\_key, None)  
 return self  
   
 def get(self, key: str) -> str:  
 """Get style value"""  
 css\_key = key.replace('\_', '-')  
 return self.rules.get(css\_key, '')  
   
 def to\_dict(self) -> Dict[str, str]:  
 """Convert to dictionary"""  
 return self.rules.copy()  
   
 def to\_string(self) -> str:  
 """Convert to CSS string"""  
 try:  
 return '; '.join(f'{k}: {v}' for k, v in self.rules.items())  
 except Exception as e:  
 raise StyleError(f"Failed to convert style to string: {e}") from e  
   
 def to\_class\_string(self) -> str:  
 """Convert to CSS class definition"""  
 return ' '.join(self.rules.keys())  
   
 def inline(self) -> str:  
 """Convert to inline style string"""  
 return self.to\_string()  
   
 def update(self: T, \*\*styles: Any) -> T:  
 """Update CSS styles"""  
 return self.add(\*\*styles)  
   
 def merge(self: T, other: Style) -> T:  
 """Merge with another style"""  
 if not isinstance(other, Style):  
 raise TypeError("Can only merge with another Style object")  
 new\_style = self.\_\_class\_\_()  
 new\_style.rules.update(self.rules)  
 new\_style.rules.update(other.rules)  
 return new\_style  
   
 def clone(self: T) -> T:  
 """Create a copy of this style"""  
 new\_style = self.\_\_class\_\_()  
 new\_style.rules.update(self.rules)  
 return new\_style  
   
 def \_\_getattr\_\_(self, name: str) -> str:  
 """Get style value using attribute access"""  
 return self.get(name)  
   
 def \_\_add\_\_(self: T, other: Style) -> T:  
 """Combine two styles"""  
 return self.merge(other)  
   
 def \_\_str\_\_(self) -> str:  
 """Convert to string"""  
 return self.to\_string()  
  
class StyleSystem:  
 """Modern styling system with advanced features"""  
   
 @staticmethod  
 def create\_gradient(start\_color: str, end\_color: str, direction: str = "to right") -> str:  
 """Create linear gradient"""  
 return f"linear-gradient({direction}, {start\_color}, {end\_color})"  
   
 @staticmethod  
 def create\_glass\_effect(opacity: float = 0.1) -> Dict[str, str]:  
 """Create glass morphism effect"""  
 return {  
 "background": f"rgba(255, 255, 255, {opacity})",  
 "backdrop\_filter": "blur(10px)",  
 "border": "1px solid rgba(255, 255, 255, 0.2)",  
 "box\_shadow": "0 8px 32px 0 rgba(31, 38, 135, 0.37)"  
 }  
   
 @staticmethod  
 def create\_neumorphism(color: str, type: str = "flat") -> Dict[str, str]:  
 """Create neumorphism effect"""  
 if type == "pressed":  
 return {  
 "background": color,  
 "box\_shadow": f"inset 5px 5px 10px rgba(0, 0, 0, 0.1), inset -5px -5px 10px rgba(255, 255, 255, 0.1)"  
 }  
 else:  
 return {  
 "background": color,  
 "box\_shadow": "5px 5px 10px rgba(0, 0, 0, 0.1), -5px -5px 10px rgba(255, 255, 255, 0.1)"  
 }  
   
 @staticmethod  
 def create\_text\_gradient(start\_color: str, end\_color: str) -> Dict[str, str]:  
 """Create text gradient effect"""  
 return {  
 "background": f"linear-gradient(to right, {start\_color}, {end\_color})",  
 "background\_clip": "text",  
 "text\_fill\_color": "transparent",  
 "-webkit-background-clip": "text",  
 "-webkit-text-fill-color": "transparent"  
 }  
   
 @staticmethod  
 def create\_animation(keyframes: Dict[str, Dict[str, str]], duration: str = "0.3s", timing: str = "ease") -> Dict[str, str]:  
 """Create CSS animation"""  
 animation\_name = f"animation\_{hash(str(keyframes))}"  
 keyframe\_rules = []  
   
 for selector, styles in keyframes.items():  
 style\_rules = [f"{k}: {v}" for k, v in styles.items()]  
 keyframe\_rules.append(f"{selector} {{ {'; '.join(style\_rules)} }}")  
   
 keyframe\_css = f"@keyframes {animation\_name} {{ {' '.join(keyframe\_rules)} }}"  
   
 # TODO: Add keyframe CSS to global styles  
   
 return {  
 "animation": f"{animation\_name} {duration} {timing}"  
 }  
   
 @staticmethod  
 def create\_transition(properties: List[str], duration: str = "0.3s", timing: str = "ease") -> str:  
 """Create CSS transition"""  
 return ", ".join([f"{prop} {duration} {timing}" for prop in properties])  
   
 @staticmethod  
 def create\_media\_query(breakpoint: str, styles: Dict[str, str]) -> str:  
 """Create media query"""  
 return f"@media (min-width: {breakpoint}) {{ {'; '.join([f'{k}: {v}' for k, v in styles.items()])} }}"  
   
 @staticmethod  
 def create\_hover\_effect(styles: Dict[str, str]) -> Dict[str, str]:  
 """Create hover effect styles"""  
 return {f"&:hover": styles}  
   
 @staticmethod  
 def create\_focus\_effect(styles: Dict[str, str]) -> Dict[str, str]:  
 """Create focus effect styles"""  
 return {f"&:focus": styles}  
   
 @staticmethod  
 def create\_active\_effect(styles: Dict[str, str]) -> Dict[str, str]:  
 """Create active effect styles"""  
 return {f"&:active": styles}  
  
class ModernStyle(Style):  
 """Enhanced style class with modern features"""  
   
 def add\_glass\_effect(self, opacity: float = 0.1):  
 """Add glass morphism effect"""  
 self.add(\*\*StyleSystem.create\_glass\_effect(opacity))  
 return self  
   
 def add\_neumorphism(self, color: str, type: str = "flat"):  
 """Add neumorphism effect"""  
 self.add(\*\*StyleSystem.create\_neumorphism(color, type))  
 return self  
   
 def add\_text\_gradient(self, start\_color: str, end\_color: str):  
 """Add text gradient effect"""  
 self.add(\*\*StyleSystem.create\_text\_gradient(start\_color, end\_color))  
 return self  
   
 def add\_animation(self, keyframes: Dict[str, Dict[str, str]], duration: str = "0.3s", timing: str = "ease"):  
 """Add CSS animation"""  
 self.add(\*\*StyleSystem.create\_animation(keyframes, duration, timing))  
 return self  
   
 def add\_transition(self, properties: List[str], duration: str = "0.3s", timing: str = "ease"):  
 """Add CSS transition"""  
 self.add(transition=StyleSystem.create\_transition(properties, duration, timing))  
 return self  
   
 def add\_hover(self, styles: Dict[str, str]):  
 """Add hover effect"""  
 self.add(\*\*StyleSystem.create\_hover\_effect(styles))  
 return self  
   
 def add\_focus(self, styles: Dict[str, str]):  
 """Add focus effect"""  
 self.add(\*\*StyleSystem.create\_focus\_effect(styles))  
 return self  
   
 def add\_active(self, styles: Dict[str, str]):  
 """Add active effect"""  
 self.add(\*\*StyleSystem.create\_active\_effect(styles))  
 return self  
   
 def add\_responsive(self, breakpoint: str, styles: Dict[str, str]):  
 """Add responsive styles"""  
 self.add\_raw(StyleSystem.create\_media\_query(breakpoint, styles))  
 return self  
  
class StylePresets:  
 """Predefined modern style presets"""  
   
 @staticmethod  
 def button(variant: str = "primary", size: str = "md") -> Dict[str, str]:  
 """Button style preset"""  
 base\_styles = {  
 "border": "none",  
 "border\_radius": "0.375rem",  
 "font\_weight": "500",  
 "cursor": "pointer",  
 "transition": "all 0.2s ease-in-out"  
 }  
   
 # Size variants  
 sizes = {  
 "sm": {"padding": "0.5rem 1rem", "font\_size": "0.875rem"},  
 "md": {"padding": "0.75rem 1.5rem", "font\_size": "1rem"},  
 "lg": {"padding": "1rem 2rem", "font\_size": "1.125rem"}  
 }  
   
 # Color variants  
 variants = {  
 "primary": {  
 "background": "#3b82f6",  
 "color": "#ffffff",  
 "&:hover": {"background": "#2563eb"},  
 "&:active": {"background": "#1d4ed8"}  
 },  
 "secondary": {  
 "background": "#6b7280",  
 "color": "#ffffff",  
 "&:hover": {"background": "#4b5563"},  
 "&:active": {"background": "#374151"}  
 },  
 "outline": {  
 "background": "transparent",  
 "border": "2px solid #3b82f6",  
 "color": "#3b82f6",  
 "&:hover": {"background": "#3b82f6", "color": "#ffffff"},  
 "&:active": {"background": "#2563eb", "color": "#ffffff"}  
 },  
 "ghost": {  
 "background": "transparent",  
 "color": "#3b82f6",  
 "&:hover": {"background": "rgba(59, 130, 246, 0.1)"},  
 "&:active": {"background": "rgba(59, 130, 246, 0.2)"}  
 }  
 }  
   
 return {\*\*base\_styles, \*\*sizes[size], \*\*variants[variant]}  
   
 @staticmethod  
 def card(elevation: str = "md") -> Dict[str, str]:  
 """Card style preset"""  
 base\_styles = {  
 "background": "#ffffff",  
 "border\_radius": "0.5rem",  
 "padding": "1.5rem",  
 "transition": "all 0.2s ease-in-out"  
 }  
   
 elevations = {  
 "sm": {"box\_shadow": "0 1px 2px 0 rgba(0, 0, 0, 0.05)"},  
 "md": {"box\_shadow": "0 4px 6px -1px rgba(0, 0, 0, 0.1)"},  
 "lg": {"box\_shadow": "0 10px 15px -3px rgba(0, 0, 0, 0.1)"}  
 }  
   
 return {\*\*base\_styles, \*\*elevations[elevation]}  
   
 @staticmethod  
 def input(variant: str = "outline") -> Dict[str, str]:  
 """Input style preset"""  
 base\_styles = {  
 "padding": "0.75rem 1rem",  
 "font\_size": "1rem",  
 "border\_radius": "0.375rem",  
 "transition": "all 0.2s ease-in-out",  
 "&:focus": {  
 "outline": "none",  
 "ring": "2px",  
 "ring\_color": "rgba(59, 130, 246, 0.5)"  
 }  
 }  
   
 variants = {  
 "outline": {  
 "border": "1px solid #d1d5db",  
 "background": "#ffffff",  
 "&:hover": {"border\_color": "#9ca3af"},  
 "&:focus": {"border\_color": "#3b82f6"}  
 },  
 "filled": {  
 "border": "1px solid transparent",  
 "background": "#f3f4f6",  
 "&:hover": {"background": "#e5e7eb"},  
 "&:focus": {"background": "#ffffff", "border\_color": "#3b82f6"}  
 },  
 "flushed": {  
 "border": "none",  
 "border\_bottom": "1px solid #d1d5db",  
 "border\_radius": "0",  
 "&:hover": {"border\_bottom\_color": "#9ca3af"},  
 "&:focus": {"border\_bottom\_color": "#3b82f6"}  
 }  
 }  
   
 return {\*\*base\_styles, \*\*variants[variant]}  
   
 @staticmethod  
 def badge(variant: str = "primary") -> Dict[str, str]:  
 """Badge style preset"""  
 base\_styles = {  
 "display": "inline-flex",  
 "align\_items": "center",  
 "padding": "0.25rem 0.75rem",  
 "font\_size": "0.875rem",  
 "font\_weight": "500",  
 "border\_radius": "9999px",  
 "line\_height": "1"  
 }  
   
 variants = {  
 "primary": {  
 "background": "#e0f2fe",  
 "color": "#0369a1"  
 },  
 "success": {  
 "background": "#dcfce7",  
 "color": "#15803d"  
 },  
 "warning": {  
 "background": "#fff3e0",  
 "color": "#ef6c00"  
 },  
 "error": {  
 "background": "#fee2e2",  
 "color": "#b91c1c"  
 }  
 }  
   
 return {\*\*base\_styles, \*\*variants[variant]}  
  
# Helper functions for creating style units  
def px(value: Union[int, float]) -> StyleUnit:  
 """Create pixel unit"""  
 return StyleUnit(value, 'px')  
  
def em(value: Union[int, float]) -> StyleUnit:  
 """Create em unit"""  
 return StyleUnit(value, 'em')  
  
def rem(value: Union[int, float]) -> StyleUnit:  
 """Create rem unit"""  
 return StyleUnit(value, 'rem')  
  
def percent(value: Union[int, float]) -> StyleUnit:  
 """Create percentage unit"""  
 return StyleUnit(value, '%')  
  
def vh(value: Union[int, float]) -> StyleUnit:  
 """Create viewport height unit"""  
 return StyleUnit(value, 'vh')  
  
def vw(value: Union[int, float]) -> StyleUnit:  
 """Create viewport width unit"""  
 return StyleUnit(value, 'vw')  
  
# Predefined styles  
class Styles:  
 """Predefined styles collection"""  
   
 @staticmethod  
 def flex(direction: str = 'row', justify: str = 'flex-start', align: str = 'stretch',  
 wrap: bool = False) -> Style:  
 """Create flex container style"""  
 return Style(  
 display='flex',  
 flex\_direction=direction,  
 justify\_content=justify,  
 align\_items=align,  
 flex\_wrap='wrap' if wrap else 'nowrap'  
 )  
   
 @staticmethod  
 def grid(columns: int = 12, gap: Union[str, StyleUnit] = px(16)) -> Style:  
 """Create grid container style"""  
 return Style(  
 display='grid',  
 grid\_template\_columns=f'repeat({columns}, 1fr)',  
 gap=str(gap)  
 )  
   
 @staticmethod  
 def card(shadow: bool = True, radius: Union[str, StyleUnit] = px(4)) -> Style:  
 """Create card style"""  
 style = Style(  
 padding=px(16),  
 border\_radius=str(radius),  
 background\_color='#ffffff'  
 )  
 if shadow:  
 style.add(box\_shadow='0 2px 4px rgba(0,0,0,0.1)')  
 return style  
   
 @staticmethod  
 def button(variant: str = 'primary') -> Style:  
 """Create button style"""  
 base\_style = Style(  
 padding=f'{px(8)} {px(16)}',  
 border\_radius=px(4),  
 border='none',  
 cursor='pointer',  
 font\_weight='500',  
 transition='all 0.2s ease'  
 )  
   
 variants = {  
 'primary': Style(  
 background\_color='#1976d2',  
 color='#ffffff',  
 hover={'background\_color': '#1565c0'}  
 ),  
 'secondary': Style(  
 background\_color='#9e9e9e',  
 color='#ffffff',  
 hover={'background\_color': '#757575'}  
 ),  
 'outlined': Style(  
 background\_color='transparent',  
 color='#1976d2',  
 border='1px solid #1976d2',  
 hover={'background\_color': 'rgba(25,118,210,0.04)'}  
 )  
 }  
   
 return base\_style + variants.get(variant, variants['primary'])  
  
# Default styles  
DEFAULT\_STYLES = """  
.pytoweb-container {  
 width: 100%;  
 margin: 0 auto;  
 padding: 0 16px;  
 box-sizing: border-box;  
}  
  
.pytoweb-row {  
 display: flex;  
 flex-wrap: wrap;  
 margin: 0 -8px;  
}  
  
.pytoweb-col {  
 padding: 0 8px;  
 box-sizing: border-box;  
}  
  
.pytoweb-card {  
 background: #ffffff;  
 border-radius: 4px;  
 padding: 16px;  
 box-shadow: 0 2px 4px rgba(0,0,0,0.1);  
}  
  
.pytoweb-button {  
 display: inline-block;  
 padding: 8px 16px;  
 border: none;  
 border-radius: 4px;  
 cursor: pointer;  
 font-weight: 500;  
 text-align: center;  
 transition: all 0.2s ease;  
}  
  
.pytoweb-button:hover {  
 opacity: 0.9;  
}  
  
.pytoweb-input {  
 width: 100%;  
 padding: 8px;  
 border: 1px solid #ddd;  
 border-radius: 4px;  
 box-sizing: border-box;  
}  
  
.pytoweb-input:focus {  
 outline: none;  
 border-color: #1976d2;  
}  
  
.pytoweb-label {  
 display: block;  
 margin-bottom: 8px;  
 font-weight: 500;  
}  
  
.pytoweb-select {  
 width: 100%;  
 padding: 8px;  
 border: 1px solid #ddd;  
 border-radius: 4px;  
 background-color: #ffffff;  
 cursor: pointer;  
}  
  
.pytoweb-checkbox {  
 margin-right: 8px;  
}  
  
.pytoweb-radio {  
 margin-right: 8px;  
}  
  
.pytoweb-textarea {  
 width: 100%;  
 padding: 8px;  
 border: 1px solid #ddd;  
 border-radius: 4px;  
 min-height: 100px;  
 resize: vertical;  
}  
  
.pytoweb-form {  
 width: 100%;  
}  
  
.pytoweb-form-group {  
 margin-bottom: 16px;  
}  
  
.pytoweb-alert {  
 padding: 12px;  
 border-radius: 4px;  
 margin-bottom: 16px;  
}  
  
.pytoweb-alert-success {  
 background-color: #e8f5e9;  
 color: #2e7d32;  
}  
  
.pytoweb-alert-error {  
 background-color: #ffebee;  
 color: #c62828;  
}  
  
.pytoweb-alert-warning {  
 background-color: #fff3e0;  
 color: #ef6c00;  
}  
  
.pytoweb-alert-info {  
 background-color: #e3f2fd;  
 color: #1565c0;  
}  
"""

themes.py

"""  
Modern theme system with design tokens and variants  
"""  
  
from typing import Dict, Any  
  
class Theme:  
 """Theme management class"""  
   
 def \_\_init\_\_(self, name: str = "default"):  
 self.name = name  
 self.tokens = {  
 # Color System  
 "colors": {  
 # Brand Colors  
 "primary": {  
 "main": "#1976d2",  
 "light": "#42a5f5",  
 "dark": "#1565c0",  
 "contrast": "#ffffff"  
 },  
 "secondary": {  
 "main": "#9c27b0",  
 "light": "#ba68c8",  
 "dark": "#7b1fa2",  
 "contrast": "#ffffff"  
 },  
 # Semantic Colors  
 "success": {  
 "main": "#2e7d32",  
 "light": "#4caf50",  
 "dark": "#1b5e20",  
 "contrast": "#ffffff"  
 },  
 "warning": {  
 "main": "#ed6c02",  
 "light": "#ff9800",  
 "dark": "#e65100",  
 "contrast": "#ffffff"  
 },  
 "error": {  
 "main": "#d32f2f",  
 "light": "#ef5350",  
 "dark": "#c62828",  
 "contrast": "#ffffff"  
 },  
 "info": {  
 "main": "#0288d1",  
 "light": "#03a9f4",  
 "dark": "#01579b",  
 "contrast": "#ffffff"  
 },  
 # Gray Scale  
 "gray": {  
 "50": "#fafafa",  
 "100": "#f5f5f5",  
 "200": "#eeeeee",  
 "300": "#e0e0e0",  
 "400": "#bdbdbd",  
 "500": "#9e9e9e",  
 "600": "#757575",  
 "700": "#616161",  
 "800": "#424242",  
 "900": "#212121"  
 },  
 # Background & Surface  
 "background": {  
 "default": "#ffffff",  
 "paper": "#ffffff",  
 "alt": "#f8f9fa"  
 },  
 # Text Colors  
 "text": {  
 "primary": "rgba(0, 0, 0, 0.87)",  
 "secondary": "rgba(0, 0, 0, 0.6)",  
 "disabled": "rgba(0, 0, 0, 0.38)",  
 "hint": "rgba(0, 0, 0, 0.38)"  
 }  
 },  
   
 # Typography System  
 "typography": {  
 "fontFamily": {  
 "primary": "'Inter', -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif",  
 "code": "'Fira Code', 'Consolas', 'Monaco', 'Andale Mono', monospace"  
 },  
 "fontWeight": {  
 "light": 300,  
 "regular": 400,  
 "medium": 500,  
 "semibold": 600,  
 "bold": 700  
 },  
 "fontSize": {  
 "xs": "0.75rem",  
 "sm": "0.875rem",  
 "base": "1rem",  
 "lg": "1.125rem",  
 "xl": "1.25rem",  
 "2xl": "1.5rem",  
 "3xl": "1.875rem",  
 "4xl": "2.25rem",  
 "5xl": "3rem"  
 },  
 "lineHeight": {  
 "none": 1,  
 "tight": 1.25,  
 "snug": 1.375,  
 "normal": 1.5,  
 "relaxed": 1.625,  
 "loose": 2  
 },  
 "letterSpacing": {  
 "tighter": "-0.05em",  
 "tight": "-0.025em",  
 "normal": "0",  
 "wide": "0.025em",  
 "wider": "0.05em",  
 "widest": "0.1em"  
 }  
 },  
   
 # Spacing System  
 "spacing": {  
 "0": "0",  
 "1": "0.25rem",  
 "2": "0.5rem",  
 "3": "0.75rem",  
 "4": "1rem",  
 "5": "1.25rem",  
 "6": "1.5rem",  
 "8": "2rem",  
 "10": "2.5rem",  
 "12": "3rem",  
 "16": "4rem",  
 "20": "5rem",  
 "24": "6rem",  
 "32": "8rem"  
 },  
   
 # Border System  
 "borders": {  
 "width": {  
 "none": "0",  
 "thin": "1px",  
 "medium": "2px",  
 "thick": "4px"  
 },  
 "radius": {  
 "none": "0",  
 "sm": "0.125rem",  
 "base": "0.25rem",  
 "md": "0.375rem",  
 "lg": "0.5rem",  
 "xl": "0.75rem",  
 "2xl": "1rem",  
 "full": "9999px"  
 },  
 "style": {  
 "solid": "solid",  
 "dashed": "dashed",  
 "dotted": "dotted"  
 }  
 },  
   
 # Shadow System  
 "shadows": {  
 "none": "none",  
 "sm": "0 1px 2px 0 rgba(0, 0, 0, 0.05)",  
 "base": "0 1px 3px 0 rgba(0, 0, 0, 0.1), 0 1px 2px 0 rgba(0, 0, 0, 0.06)",  
 "md": "0 4px 6px -1px rgba(0, 0, 0, 0.1), 0 2px 4px -1px rgba(0, 0, 0, 0.06)",  
 "lg": "0 10px 15px -3px rgba(0, 0, 0, 0.1), 0 4px 6px -2px rgba(0, 0, 0, 0.05)",  
 "xl": "0 20px 25px -5px rgba(0, 0, 0, 0.1), 0 10px 10px -5px rgba(0, 0, 0, 0.04)",  
 "2xl": "0 25px 50px -12px rgba(0, 0, 0, 0.25)",  
 "inner": "inset 0 2px 4px 0 rgba(0, 0, 0, 0.06)"  
 },  
   
 # Animation System  
 "animation": {  
 "duration": {  
 "fastest": "75ms",  
 "faster": "100ms",  
 "fast": "150ms",  
 "normal": "200ms",  
 "slow": "300ms",  
 "slower": "400ms",  
 "slowest": "500ms"  
 },  
 "easing": {  
 "linear": "linear",  
 "ease": "ease",  
 "easeIn": "cubic-bezier(0.4, 0, 1, 1)",  
 "easeOut": "cubic-bezier(0, 0, 0.2, 1)",  
 "easeInOut": "cubic-bezier(0.4, 0, 0.2, 1)"  
 }  
 },  
   
 # Z-index System  
 "zIndex": {  
 "hide": -1,  
 "base": 0,  
 "dropdown": 1000,  
 "sticky": 1100,  
 "fixed": 1200,  
 "modalBackdrop": 1300,  
 "modal": 1400,  
 "popover": 1500,  
 "tooltip": 1600  
 },  
   
 # Breakpoints System  
 "breakpoints": {  
 "xs": "0px",  
 "sm": "600px",  
 "md": "900px",  
 "lg": "1200px",  
 "xl": "1536px"  
 },  
   
 # Grid System  
 "grid": {  
 "columns": 12,  
 "gutter": {  
 "xs": "1rem",  
 "sm": "1.5rem",  
 "md": "2rem",  
 "lg": "2.5rem",  
 "xl": "3rem"  
 },  
 "margin": {  
 "xs": "1rem",  
 "sm": "1.5rem",  
 "md": "2rem",  
 "lg": "2.5rem",  
 "xl": "3rem"  
 }  
 }  
 }  
   
 def get\_token(self, path: str) -> Any:  
 """Get design token value by path"""  
 keys = path.split(".")  
 value = self.tokens  
 for key in keys:  
 value = value.get(key)  
 if value is None:  
 return None  
 return value  
   
 def set\_token(self, path: str, value: Any):  
 """Set design token value by path"""  
 keys = path.split(".")  
 target = self.tokens  
 for key in keys[:-1]:  
 target = target.setdefault(key, {})  
 target[keys[-1]] = value  
   
 def create\_variant(self, name: str, overrides: Dict[str, Any]) -> "Theme":  
 """Create theme variant with overrides"""  
 variant = Theme(f"{self.name}-{name}")  
 variant.tokens = self.tokens.copy()  
   
 for path, value in overrides.items():  
 variant.set\_token(path, value)  
   
 return variant  
   
 @classmethod  
 def create\_dark\_theme(cls) -> "Theme":  
 """Create dark theme variant"""  
 dark\_theme = cls("dark")  
 dark\_theme.tokens["colors"].update({  
 "background": {  
 "default": "#121212",  
 "paper": "#1e1e1e",  
 "alt": "#2c2c2c"  
 },  
 "text": {  
 "primary": "rgba(255, 255, 255, 0.87)",  
 "secondary": "rgba(255, 255, 255, 0.6)",  
 "disabled": "rgba(255, 255, 255, 0.38)",  
 "hint": "rgba(255, 255, 255, 0.38)"  
 }  
 })  
 return dark\_theme  
   
 @classmethod  
 def create\_high\_contrast\_theme(cls) -> "Theme":  
 """Create high contrast theme variant"""  
 high\_contrast = cls("high-contrast")  
 high\_contrast.tokens["colors"].update({  
 "background": {  
 "default": "#000000",  
 "paper": "#000000",  
 "alt": "#1a1a1a"  
 },  
 "text": {  
 "primary": "#ffffff",  
 "secondary": "#ffffff",  
 "disabled": "#808080",  
 "hint": "#808080"  
 }  
 })  
 return high\_contrast  
  
class ThemeProvider:  
 """Theme provider for components"""  
   
 \_current\_theme: Theme = None  
   
 @classmethod  
 def set\_theme(cls, theme: Theme):  
 """Set current theme"""  
 cls.\_current\_theme = theme  
   
 @classmethod  
 def get\_theme(cls) -> Theme:  
 """Get current theme"""  
 return cls.\_current\_theme  
   
 @classmethod  
 def use\_theme(cls) -> Theme:  
 """Get current theme or default"""  
 if cls.\_current\_theme is None:  
 cls.\_current\_theme = Theme({})  
 return cls.\_current\_theme

validation.py

"""Form validation system for PytoWeb"""  
from typing import Dict, Any, List, Optional, Callable  
import re  
  
class ValidationRule:  
 """Base class for validation rules"""  
 def \_\_init\_\_(self, message: str):  
 self.message = message  
   
 def validate(self, value: Any) -> bool:  
 raise NotImplementedError  
  
class Required(ValidationRule):  
 """Required field validation"""  
 def \_\_init\_\_(self, message: str = "This field is required"):  
 super().\_\_init\_\_(message)  
   
 def validate(self, value: Any) -> bool:  
 if value is None:  
 return False  
 if isinstance(value, str) and not value.strip():  
 return False  
 return True  
  
class MinLength(ValidationRule):  
 """Minimum length validation"""  
 def \_\_init\_\_(self, min\_length: int, message: str = None):  
 super().\_\_init\_\_(message or f"Minimum length is {min\_length}")  
 self.min\_length = min\_length  
   
 def validate(self, value: str) -> bool:  
 return len(str(value)) >= self.min\_length  
  
class MaxLength(ValidationRule):  
 """Maximum length validation"""  
 def \_\_init\_\_(self, max\_length: int, message: str = None):  
 super().\_\_init\_\_(message or f"Maximum length is {max\_length}")  
 self.max\_length = max\_length  
   
 def validate(self, value: str) -> bool:  
 return len(str(value)) <= self.max\_length  
  
class Pattern(ValidationRule):  
 """Pattern validation using regex"""  
 def \_\_init\_\_(self, pattern: str, message: str = "Invalid format"):  
 super().\_\_init\_\_(message)  
 self.pattern = re.compile(pattern)  
   
 def validate(self, value: str) -> bool:  
 return bool(self.pattern.match(str(value)))  
  
class Email(ValidationRule):  
 """Email format validation"""  
 def \_\_init\_\_(self, message: str = "Invalid email format"):  
 super().\_\_init\_\_(message)  
 self.pattern = re.compile(r"^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")  
   
 def validate(self, value: str) -> bool:  
 return bool(self.pattern.match(str(value)))  
  
class Range(ValidationRule):  
 """Numeric range validation"""  
 def \_\_init\_\_(self, min\_value: float = None, max\_value: float = None, message: str = None):  
 message = message or f"Value must be between {min\_value} and {max\_value}"  
 super().\_\_init\_\_(message)  
 self.min\_value = min\_value  
 self.max\_value = max\_value  
   
 def validate(self, value: float) -> bool:  
 try:  
 num = float(value)  
 if self.min\_value is not None and num < self.min\_value:  
 return False  
 if self.max\_value is not None and num > self.max\_value:  
 return False  
 return True  
 except (TypeError, ValueError):  
 return False  
  
class Custom(ValidationRule):  
 """Custom validation using a callback function"""  
 def \_\_init\_\_(self, validator: Callable[[Any], bool], message: str):  
 super().\_\_init\_\_(message)  
 self.validator = validator  
   
 def validate(self, value: Any) -> bool:  
 return self.validator(value)  
  
class FormValidator:  
 """Form validation manager"""  
 def \_\_init\_\_(self):  
 self.fields: Dict[str, List[ValidationRule]] = {}  
 self.errors: Dict[str, List[str]] = {}  
   
 def add\_field(self, field\_name: str, rules: List[ValidationRule]):  
 """Add validation rules for a field"""  
 self.fields[field\_name] = rules  
   
 def validate(self, data: Dict[str, Any]) -> bool:  
 """Validate form data"""  
 self.errors.clear()  
 is\_valid = True  
   
 for field\_name, rules in self.fields.items():  
 field\_value = data.get(field\_name)  
 field\_errors = []  
   
 for rule in rules:  
 if not rule.validate(field\_value):  
 field\_errors.append(rule.message)  
 is\_valid = False  
   
 if field\_errors:  
 self.errors[field\_name] = field\_errors  
   
 return is\_valid  
   
 def get\_errors(self) -> Dict[str, List[str]]:  
 """Get validation errors"""  
 return self.errors

vdom.py

"""Virtual DOM implementation for PytoWeb."""  
from typing import Dict, List, Optional, Any  
import difflib  
  
class VNode:  
 """Virtual DOM Node."""  
 def \_\_init\_\_(self, tag: str, props: Dict = None, children: List = None):  
 self.tag = tag  
 self.props = props or {}  
 self.children = children or []  
 self.key = props.get('key') if props else None  
   
 def \_\_eq\_\_(self, other):  
 if not isinstance(other, VNode):  
 return False  
 return (self.tag == other.tag and   
 self.props == other.props and   
 self.children == other.children)  
  
class VDOMDiffer:  
 """Handles virtual DOM diffing and patching."""  
   
 @staticmethod  
 def diff(old\_node: Optional[VNode], new\_node: Optional[VNode]) -> List[Dict]:  
 """Generate a list of patches based on differences between nodes."""  
 patches = []  
   
 if old\_node is None:  
 patches.append({  
 'type': 'CREATE',  
 'node': new\_node  
 })  
 elif new\_node is None:  
 patches.append({  
 'type': 'REMOVE'  
 })  
 elif old\_node != new\_node:  
 if old\_node.tag != new\_node.tag:  
 patches.append({  
 'type': 'REPLACE',  
 'node': new\_node  
 })  
 else:  
 # Props diff  
 props\_patch = VDOMDiffer.\_diff\_props(old\_node.props, new\_node.props)  
 if props\_patch:  
 patches.append({  
 'type': 'PROPS',  
 'props': props\_patch  
 })  
   
 # Children diff  
 children\_patches = VDOMDiffer.\_diff\_children(  
 old\_node.children,  
 new\_node.children  
 )  
 patches.extend(children\_patches)  
   
 return patches  
   
 @staticmethod  
 def \_diff\_props(old\_props: Dict, new\_props: Dict) -> Optional[Dict]:  
 """Compare props and return differences."""  
 props\_patch = {}  
   
 # Check for changed or new props  
 for key, value in new\_props.items():  
 if key not in old\_props or old\_props[key] != value:  
 props\_patch[key] = value  
   
 # Check for removed props  
 for key in old\_props:  
 if key not in new\_props:  
 props\_patch[key] = None  
   
 return props\_patch if props\_patch else None  
   
 @staticmethod  
 def \_diff\_children(old\_children: List[VNode], new\_children: List[VNode]) -> List[Dict]:  
 """Compare children nodes and return patches."""  
 patches = []  
   
 # Use difflib for optimal diff  
 matcher = difflib.SequenceMatcher(None, old\_children, new\_children)  
 for tag, i1, i2, j1, j2 in matcher.get\_opcodes():  
 if tag == 'replace':  
 for i in range(i1, i2):  
 patches.append({  
 'type': 'REPLACE\_CHILD',  
 'index': i,  
 'node': new\_children[j1 + (i - i1)] if i - i1 < j2 - j1 else None  
 })  
 elif tag == 'delete':  
 for i in range(i1, i2):  
 patches.append({  
 'type': 'REMOVE\_CHILD',  
 'index': i  
 })  
 elif tag == 'insert':  
 for j in range(j1, j2):  
 patches.append({  
 'type': 'INSERT\_CHILD',  
 'index': j,  
 'node': new\_children[j]  
 })  
   
 return patches  
  
class VDOMRenderer:  
 """Handles rendering virtual DOM to real DOM."""  
 \_string\_pool = {}  
 \_pool\_size = 1000  
   
 @staticmethod  
 def \_get\_pooled\_string(s: str) -> str:  
 if s not in VDOMRenderer.\_string\_pool:  
 if len(VDOMRenderer.\_string\_pool) >= VDOMRenderer.\_pool\_size:  
 VDOMRenderer.\_string\_pool.clear()  
 VDOMRenderer.\_string\_pool[s] = s  
 return VDOMRenderer.\_string\_pool[s]  
   
 @staticmethod  
 def create\_element(vnode: VNode) -> str:  
 if isinstance(vnode, str):  
 return VDOMRenderer.\_get\_pooled\_string(vnode)  
   
 void\_elements = {  
 'area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input',  
 'link', 'meta', 'param', 'source', 'track', 'wbr'  
 }  
   
 tag = VDOMRenderer.\_get\_pooled\_string(vnode.tag)  
 html = ['<{}'.format(tag)]  
   
 if vnode.props:  
 props\_str = VDOMRenderer.\_props\_to\_string(vnode.props)  
 html.append(VDOMRenderer.\_get\_pooled\_string(props\_str))  
   
 if tag in void\_elements:  
 html.append('/>')  
 return VDOMRenderer.\_get\_pooled\_string(''.join(html))  
   
 html.append('>')  
   
 if vnode.children:  
 for child in vnode.children:  
 if isinstance(child, VNode):  
 html.append(VDOMRenderer.create\_element(child))  
 else:  
 html.append(str(child))  
   
 html.append('</{}>'.format(tag))  
 return VDOMRenderer.\_get\_pooled\_string(''.join(html))  
   
 @staticmethod  
 def \_props\_to\_string(props: Dict) -> str:  
 """Convert props dictionary to HTML attributes string."""  
 if not props:  
 return ''  
   
 attributes = []  
 for key, value in props.items():  
 if value is None or value is False:  
 continue  
 if value is True:  
 attributes.append(key)  
 else:  
 # 处理事件处理器  
 if key.startswith('on'):  
 # 将Python函数转换为JavaScript事件处理器  
 value = "pytoWeb.handleEvent('{}', this)".format(key)  
 # 处理样式对象  
 elif key == 'style' and isinstance(value, dict):  
 value = ';'.join('{}:{}'.format(k, v) for k, v in value.items())  
 # 处理类名列表  
 elif key == 'class' and isinstance(value, (list, set)):  
 value = ' '.join(value)  
   
 attributes.append('{}="{}"'.format(key, str(value).replace('"', "&quot;")))  
   
 return ' ' + ' '.join(attributes) if attributes else ''  
  
 @staticmethod  
 def render\_to\_string(vnode: Any) -> str:  
 """Render a virtual DOM node to HTML string."""  
 try:  
 if isinstance(vnode, (str, int, float)):  
 return str(vnode)  
 elif isinstance(vnode, VNode):  
 return VDOMRenderer.create\_element(vnode)  
 elif hasattr(vnode, 'render'):  
 # Handle components that have a render method  
 rendered = vnode.render()  
 return VDOMRenderer.render\_to\_string(rendered)  
 elif isinstance(vnode, (list, tuple)):  
 # Handle lists of nodes  
 return ''.join(VDOMRenderer.render\_to\_string(child) for child in vnode)  
 else:  
 return str(vnode)  
 except Exception as e:  
 raise Exception(f"Failed to render node: {e}")

workers.py

"""Web Worker support for PytoWeb."""  
import json  
import threading  
import queue  
import logging  
import traceback  
from typing import Dict, Any, Callable, Optional  
from dataclasses import dataclass  
from concurrent.futures import ThreadPoolExecutor  
  
@dataclass  
class WorkerMessage:  
 """Message passed between main thread and worker."""  
 type: str  
 data: Any  
 id: Optional[str] = None  
  
class PythonWorker:  
 """Python-based worker implementation."""  
   
 def \_\_init\_\_(self, name: str):  
 self.name = name  
 self.\_running = False  
 self.\_thread: Optional[threading.Thread] = None  
 self.\_message\_queue = queue.Queue()  
 self.\_callbacks: Dict[str, Callable] = {}  
 self.\_error\_handler: Optional[Callable] = None  
   
 def start(self):  
 """Start the worker thread."""  
 if self.\_running:  
 return  
   
 self.\_running = True  
 self.\_thread = threading.Thread(target=self.\_run)  
 self.\_thread.daemon = True  
 self.\_thread.start()  
   
 def stop(self):  
 """Stop the worker thread."""  
 self.\_running = False  
 if self.\_thread:  
 self.\_thread.join()  
   
 def post\_message(self, message\_type: str, data: Any, message\_id: Optional[str] = None):  
 """Send a message to the worker."""  
 message = WorkerMessage(type=message\_type, data=data, id=message\_id)  
 self.\_message\_queue.put(message)  
   
 def on\_message(self, message\_type: str, callback: Callable):  
 """Register a message handler."""  
 self.\_callbacks[message\_type] = callback  
   
 def on\_error(self, handler: Callable):  
 """Register an error handler."""  
 self.\_error\_handler = handler  
   
 def \_run(self):  
 """Main worker loop."""  
 while self.\_running:  
 try:  
 message = self.\_message\_queue.get(timeout=1.0)  
 self.\_handle\_message(message)  
 except queue.Empty:  
 continue  
 except Exception as e:  
 if self.\_error\_handler:  
 self.\_error\_handler(e)  
 else:  
 logging.error(f"Worker error: {str(e)}\n{traceback.format\_exc()}")  
   
 def \_handle\_message(self, message: WorkerMessage):  
 """Handle a received message."""  
 if message.type in self.\_callbacks:  
 try:  
 result = self.\_callbacks[message.type](message.data)  
 if message.id:  
 # 如果消息有ID，发送响应  
 self.post\_message('response', {  
 'id': message.id,  
 'result': result  
 })  
 except Exception as e:  
 if message.id:  
 # 发送错误响应  
 self.post\_message('error', {  
 'id': message.id,  
 'error': str(e)  
 })  
 raise  
  
class WorkerPool:  
 """Manages a pool of workers."""  
   
 def \_\_init\_\_(self, size: int = 4):  
 self.\_workers: Dict[str, PythonWorker] = {}  
 self.\_executor = ThreadPoolExecutor(max\_workers=size)  
 self.\_size = size  
   
 def create\_worker(self, name: str) -> PythonWorker:  
 """Create a new worker."""  
 if name in self.\_workers:  
 raise ValueError(f"Worker '{name}' already exists")  
   
 worker = PythonWorker(name)  
 self.\_workers[name] = worker  
 worker.start()  
 return worker  
   
 def get\_worker(self, name: str) -> Optional[PythonWorker]:  
 """Get an existing worker."""  
 return self.\_workers.get(name)  
   
 def remove\_worker(self, name: str):  
 """Remove and stop a worker."""  
 if name in self.\_workers:  
 worker = self.\_workers.pop(name)  
 worker.stop()  
   
 def stop\_all(self):  
 """Stop all workers."""  
 for worker in self.\_workers.values():  
 worker.stop()  
 self.\_workers.clear()  
 self.\_executor.shutdown()  
  
class WorkerDecorators:  
 """Decorators for worker functionality."""  
   
 @staticmethod  
 def run\_in\_worker(worker\_name: str):  
 """Decorator to run a function in a worker."""  
 def decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 worker = WorkerPool().get\_worker(worker\_name)  
 if not worker:  
 worker = WorkerPool().create\_worker(worker\_name)  
   
 # 创建消息ID  
 message\_id = f"{func.\_\_name\_\_}\_{id(args)}\_{id(kwargs)}"  
   
 # 创建Future对象  
 future = threading.Event()  
 result = {'value': None, 'error': None}  
   
 def handle\_response(data):  
 if data['id'] == message\_id:  
 if 'result' in data:  
 result['value'] = data['result']  
 else:  
 result['error'] = data['error']  
 future.set()  
   
 worker.on\_message('response', handle\_response)  
 worker.post\_message(func.\_\_name\_\_, {  
 'args': args,  
 'kwargs': kwargs  
 }, message\_id)  
   
 # 等待结果  
 future.wait()  
   
 if result['error']:  
 raise Exception(f"Error in worker: {result['error']}")  
 return result['value']  
   
 return wrapper  
 return decorator  
   
 @staticmethod  
 def worker\_method(message\_type: str):  
 """Decorator to register a worker method."""  
 def decorator(func):  
 def wrapper(self, \*args, \*\*kwargs):  
 if isinstance(self, PythonWorker):  
 return func(self, \*args, \*\*kwargs)  
 else:  
 raise TypeError("Decorator must be used with PythonWorker class")  
   
 # 注册消息处理器  
 if hasattr(func, '\_\_self\_\_') and isinstance(func.\_\_self\_\_, PythonWorker):  
 func.\_\_self\_\_.on\_message(message\_type, wrapper)  
   
 return wrapper  
 return decorator