SP.sol

```
// SPDX-License-Identifier: GNU AGPLv3
pragma solidity ^0.8.26;

import { ISP } from "../interfaces/ISP.sol";
import { ISPHook } from "../interfaces/ISPHook.sol";
import { ISPGlobalHook } from "../interfaces/ISPGlobalHook.sol";
import { Schema } from "../models/Schema.sol";
import { Attestation, OffchainAttestation } from "../models/Attestation.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SignatureChecker } from
"@openzeppelin/contracts/utils/cryptography/SignatureChecker.sol";
import { MessageHashUtils } from
"@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import { UUPSUpgradeable } from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";
import { OwnableUpgradeable } from "@openzeppelin/contracts-
upgradeable/access/OwnableUpgradeable.sol";

// solhint-disable var-name-mixedcase
contract SP is ISP, UUPSUpgradeable, OwnableUpgradeable {
  /// @custom:storage-location erc7201:ethsign.SP
  struct SPStorage {
    bool paused;
    mapping(uint64 => Schema) schemaRegistry;
    mapping(uint64 => Attestation) attestationRegistry;
    mapping(string => OffchainAttestation) offchainAttestationRegistry;
    uint64 schemaCounter;
    uint64 attestationCounter;
    uint64 initialSchemaCounter;
    uint64 initialAttestationCounter;
```

```
    ISPGlobalHook globalHook;
  }

  // keccak256(abi.encode(uint256(keccak256("ethsign.SP")) - 1)) & ~bytes32(uint256(0xff))
  bytes32 private constant SPStorageLocation =
0x9f5ee6fb062129ebe4f4f93ab4866ee289599fbb940712219d796d503e3bd400;

  bytes32 private constant REGISTER_ACTION_NAME = "REGISTER";
  bytes32 private constant REGISTER_BATCH_ACTION_NAME = "REGISTER_BATCH";
  bytes32 private constant ATTEST_ACTION_NAME = "ATTEST";
  bytes32 private constant ATTEST_BATCH_ACTION_NAME = "ATTEST_BATCH";
  bytes32 private constant ATTEST_OFFCHAIN_ACTION_NAME = "ATTEST_OFFCHAIN";
  bytes32 private constant ATTEST_OFFCHAIN_BATCH_ACTION_NAME =
"ATTEST_OFFCHAIN_BATCH";
  bytes32 private constant REVOKE_ACTION_NAME = "REVOKE";
  bytes32 private constant REVOKE_BATCH_ACTION_NAME = "REVOKE_BATCH";
  bytes32 private constant REVOKE_OFFCHAIN_ACTION_NAME = "REVOKE_OFFCHAIN";
  bytes32 private constant REVOKE_OFFCHAIN_BATCH_ACTION_NAME =
"REVOKE_OFFCHAIN_BATCH";

  function _getSPStorage() internal pure returns (SPStorage storage $) {
    assembly {
      $.slot := SPStorageLocation
    }
  }

  /// @custom:oz-upgrades-unsafe-allow constructor
  constructor() {
    if (block.chainid != 31_337) {
      _disableInitializers();
    }
  }
```

```
  function initialize(uint64 schemaCounter_, uint64 attestationCounter_) public initializer {
    SPStorage storage $ = _getSPStorage();
    __Ownable_init(_msgSender());
    $.schemaCounter = schemaCounter_;
    $.attestationCounter = attestationCounter_;
    $.initialSchemaCounter = schemaCounter_;
    $.initialAttestationCounter = attestationCounter_;
  }

  function setGlobalHook(address hook) external onlyOwner {
    _getSPStorage().globalHook = ISPGlobalHook(hook);
  }

  function setPause(bool paused) external onlyOwner {
    _getSPStorage().paused = paused;
  }

  function register(
    Schema memory schema,
    bytes calldata delegateSignature
  )
    external
    override
    returns (uint64 schemaId)
  {
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      __checkDelegationSignature(schema.registrant, getDelegatedRegisterHash(schema),
delegateSignature);
    } else {
```

```
        if (schema.registrant != _msgSender()) revert SchemaWrongRegistrant();
    }
    schemaId = _register(schema);
    _callGlobalHook();
  }

  function attest(
    Attestation calldata attestation,
    string calldata indexingKey,
    bytes calldata delegateSignature,
    bytes calldata extraData
  )
    external
    override
    returns (uint64)
  {
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      __checkDelegationSignature(attestation.attester, getDelegatedAttestHash(attestation),
delegateSignature);
    }
    (uint64 schemaId, uint64 attestationId) = _attest(attestation, indexingKey, delegateMode);
    ISPHook hook = __getResolverFromAttestationId(attestationId);
    if (address(hook) != address(0)) {
      hook.didReceiveAttestation(attestation.attester, schemaId, attestationId, extraData);
    }
    _callGlobalHook();
    return attestationId;
  }

  function attestBatch(
```

```
    Attestation[] memory attestations,
    string[] memory indexingKeys,
    bytes memory delegateSignature,
    bytes memory extraData
  )
    external
    override
    returns (uint64[] memory attestationIds)
  {
    bool delegateMode = delegateSignature.length != 0;
    address attester = attestations[0].attester;
    if (delegateMode) {
      __checkDelegationSignature(attester, getDelegatedAttestBatchHash(attestations),
delegateSignature);
    }
    attestationIds = new uint64[](attestations.length);
    for (uint256 i = 0; i < attestations.length; i++) {
      if (delegateMode && attestations[i].attester != attester) {
        revert AttestationWrongAttester();
      }
      (uint64 schemaId, uint64 attestationId) = _attest(attestations[i], indexingKeys[i],
delegateMode);
      attestationIds[i] = attestationId;
      ISPHook hook = __getResolverFromAttestationId(attestationId);
      if (address(hook) != address(0)) {
        hook.didReceiveAttestation(attestations[i].attester, schemaId, attestationId, extraData);
      }
    }
    _callGlobalHook();
  }

  function attest(
```

```
        Attestation calldata attestation,
        uint256 resolverFeesETH,
        string calldata indexingKey,
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external
        payable
        returns (uint64)
    {
        bool delegateMode = delegateSignature.length != 0;
        if (delegateMode) {
            __checkDelegationSignature(attestation.attester, getDelegatedAttestHash(attestation),
delegateSignature);
        }
        (uint64 schemaId, uint64 attestationId) = _attest(attestation, indexingKey, delegateMode);
        ISPHook hook = __getResolverFromAttestationId(attestationId);
        if (address(hook) != address(0)) {
            hook.didReceiveAttestation{ value: resolverFeesETH }(
                attestation.attester, schemaId, attestationId, extraData
            );
        }
        _callGlobalHook();
        return attestationId;
    }

    function attestBatch(
        Attestation[] memory attestations,
        uint256[] memory resolverFeesETH,
        string[] memory indexingKeys,
        bytes memory delegateSignature,
```

```
    bytes memory extraData
  )
    external
    payable
    override
    returns (uint64[] memory attestationIds)
  {
    bool delegateMode = delegateSignature.length != 0;
    address attester = attestations[0].attester;
    if (delegateMode) {
      __checkDelegationSignature(attester, getDelegatedAttestBatchHash(attestations),
delegateSignature);
    }
    attestationIds = new uint64[](attestations.length);
    for (uint256 i = 0; i < attestations.length; i++) {
      if (delegateMode && attestations[i].attester != attester) {
        revert AttestationWrongAttester();
      }
      (uint64 schemaId, uint64 attestationId) = _attest(attestations[i], indexingKeys[i],
delegateMode);
      attestationIds[i] = attestationId;
      ISPHook hook = __getResolverFromAttestationId(attestationId);
      if (address(hook) != address(0)) {
        hook.didReceiveAttestation{ value: resolverFeesETH[i] }(
          attestations[i].attester, schemaId, attestationId, extraData
        );
      }
    }
    _callGlobalHook();
  }

  function attest(
```

```
    Attestation memory attestation,
    IERC20 resolverFeesERC20Token,
    uint256 resolverFeesERC20Amount,
    string memory indexingKey,
    bytes memory delegateSignature,
    bytes memory extraData
  )
    external
    override
    returns (uint64)
  {
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      __checkDelegationSignature(attestation.attester, getDelegatedAttestHash(attestation),
delegateSignature);
    }
    (uint64 schemaId, uint64 attestationId) = _attest(attestation, indexingKey, delegateMode);
    ISPHook hook = __getResolverFromAttestationId(attestationId);
    if (address(hook) != address(0)) {
      hook.didReceiveAttestation(
        attestation.attester,
        schemaId,
        attestationId,
        resolverFeesERC20Token,
        resolverFeesERC20Amount,
        extraData
      );
    }
    _callGlobalHook();
    return attestationId;
  }
```

```
  function attestBatch(
    Attestation[] memory attestations,
    IERC20[] memory resolverFeesERC20Tokens,
    uint256[] memory resolverFeesERC20Amount,
    string[] memory indexingKeys,
    bytes memory delegateSignature,
    bytes memory extraData
  )
    external
    override
    returns (uint64[] memory attestationIds)
  {
    bool delegateMode = delegateSignature.length != 0;
    // address attester = attestations[0].attester;
    if (delegateMode) {
      __checkDelegationSignature(
        attestations[0].attester, getDelegatedAttestBatchHash(attestations), delegateSignature
      );
    }
    attestationIds = new uint64[](attestations.length);
    for (uint256 i = 0; i < attestations.length; i++) {
      if (delegateMode && attestations[i].attester != attestations[0].attester) {
        revert AttestationWrongAttester();
      }
      (uint64 schemaId, uint64 attestationId) = _attest(attestations[i], indexingKeys[i],
delegateMode);
      attestationIds[i] = attestationId;
      ISPHook hook = __getResolverFromAttestationId(attestationId);
      if (address(hook) != address(0)) {
        hook.didReceiveAttestation(
```

```
                attestations[i].attester,
                schemaId,
                attestationId,
                resolverFeesERC20Tokens[i],
                resolverFeesERC20Amount[i],
                extraData
            );
        }
    }
    _callGlobalHook();
}

function attestOffchain(
    string calldata offchainAttestationId,
    address delegateAttester,
    bytes calldata delegateSignature
)
    external
    override
{
    address attester = _msgSender();
    if (delegateSignature.length != 0) {
        __checkDelegationSignature(
            delegateAttester, getDelegatedOffchainAttestHash(offchainAttestationId),
delegateSignature
        );
        attester = delegateAttester;
    }
    _attestOffchain(offchainAttestationId, attester);
    _callGlobalHook();
}
```

```
  function attestOffchainBatch(
    string[] calldata attestationIds,
    address delegateAttester,
    bytes calldata delegateSignature
  )
    external
    override
  {
    address attester = _msgSender();
    if (delegateSignature.length != 0) {
      __checkDelegationSignature(
        delegateAttester, getDelegatedOffchainAttestBatchHash(attestationIds),
delegateSignature
      );
      attester = delegateAttester;
    }
    for (uint256 i = 0; i < attestationIds.length; i++) {
      _attestOffchain(attestationIds[i], attester);
    }
    _callGlobalHook();
  }


  function revoke(
    uint64 attestationId,
    string calldata reason,
    bytes calldata delegateSignature,
    bytes calldata extraData
  )
    external
    override
```

```
{
    address storageAttester = _getSPStorage().attestationRegistry[attestationId].attester;
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
        __checkDelegationSignature(
            storageAttester, getDelegatedRevokeHash(attestationId, reason), delegateSignature
        );
    }
    uint64 schemaId = _revoke(attestationId, reason, delegateMode);
    ISPHook hook = __getResolverFromAttestationId(attestationId);
    if (address(hook) != address(0)) {
        hook.didReceiveRevocation(storageAttester, schemaId, attestationId, extraData);
    }
    _callGlobalHook();
}

function revokeBatch(
    uint64[] memory attestationIds,
    string[] memory reasons,
    bytes memory delegateSignature,
    bytes memory extraData
)
    external
    override
{
    address currentAttester = _msgSender();
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
        address storageAttester = _getSPStorage().attestationRegistry[attestationIds[0]].attester;
        __checkDelegationSignature(
```

```
            storageAttester, getDelegatedRevokeBatchHash(attestationIds, reasons),
delegateSignature
          );
          currentAttester = storageAttester;
        }
        for (uint256 i = 0; i < attestationIds.length; i++) {
          address storageAttester = _getSPStorage().attestationRegistry[attestationIds[i]].attester;
          if (delegateMode && storageAttester != currentAttester) {
            revert AttestationWrongAttester();
          }
          uint64 schemaId = _revoke(attestationIds[i], reasons[i], delegateMode);
          ISPHook hook = __getResolverFromAttestationId(attestationIds[i]);
          if (address(hook) != address(0)) {
            hook.didReceiveRevocation(storageAttester, schemaId, attestationIds[i], extraData);
          }
        }
        _callGlobalHook();
      }

      function revoke(
        uint64 attestationId,
        string memory reason,
        uint256 resolverFeesETH,
        bytes memory delegateSignature,
        bytes memory extraData
      )
        external
        payable
        override
      {
        address storageAttester = _getSPStorage().attestationRegistry[attestationId].attester;
```

```
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      __checkDelegationSignature(
        storageAttester, getDelegatedRevokeHash(attestationId, reason), delegateSignature
      );
    }
    uint64 schemaId = _revoke(attestationId, reason, delegateMode);
    ISPHook hook = __getResolverFromAttestationId(attestationId);
    if (address(hook) != address(0)) {
      hook.didReceiveRevocation{ value: resolverFeesETH }(storageAttester, schemaId,
attestationId, extraData);
    }
    _callGlobalHook();
  }

  function revokeBatch(
    uint64[] memory attestationIds,
    string[] memory reasons,
    uint256[] memory resolverFeesETH,
    bytes memory delegateSignature,
    bytes memory extraData
  )
    external
    payable
    override
  {
    address currentAttester = _msgSender();
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      address storageAttester = _getSPStorage().attestationRegistry[attestationIds[0]].attester;
      __checkDelegationSignature(
```

```
            storageAttester, getDelegatedRevokeBatchHash(attestationIds, reasons),
delegateSignature
        );
        currentAttester = storageAttester;
      }
      for (uint256 i = 0; i < attestationIds.length; i++) {
        address storageAttester = _getSPStorage().attestationRegistry[attestationIds[i]].attester;
        if (delegateMode && storageAttester != currentAttester) {
          revert AttestationWrongAttester();
        }
        uint64 schemaId = _revoke(attestationIds[i], reasons[i], delegateMode);
        ISPHook hook = __getResolverFromAttestationId(attestationIds[i]);
        if (address(hook) != address(0)) {
          hook.didReceiveRevocation{ value: resolverFeesETH[i] }(
            storageAttester, schemaId, attestationIds[i], extraData
          );
        }
      }
      _callGlobalHook();
    }

    function revoke(
      uint64 attestationId,
      string memory reason,
      IERC20 resolverFeesERC20Token,
      uint256 resolverFeesERC20Amount,
      bytes memory delegateSignature,
      bytes memory extraData
    )
      external
      override
```

```
  {
    address storageAttester = _getSPStorage().attestationRegistry[attestationId].attester;
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      __checkDelegationSignature(
        storageAttester, getDelegatedRevokeHash(attestationId, reason), delegateSignature
      );
    }
    uint64 schemaId = _revoke(attestationId, reason, delegateMode);
    ISPHook hook = __getResolverFromAttestationId(attestationId);
    if (address(hook) != address(0)) {
      hook.didReceiveRevocation(
        storageAttester, schemaId, attestationId, resolverFeesERC20Token,
resolverFeesERC20Amount, extraData
      );
    }
    _callGlobalHook();
  }

  function revokeBatch(
    uint64[] memory attestationIds,
    string[] memory reasons,
    IERC20[] memory resolverFeesERC20Tokens,
    uint256[] memory resolverFeesERC20Amount,
    bytes memory delegateSignature,
    bytes memory extraData
  )
    external
    override
  {
    address currentAttester = _msgSender();
```

```
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      address storageAttester = _getSPStorage().attestationRegistry[attestationIds[0]].attester;
      __checkDelegationSignature(
        storageAttester, getDelegatedRevokeBatchHash(attestationIds, reasons),
delegateSignature
      );
      currentAttester = storageAttester;
    }
    for (uint256 i = 0; i < attestationIds.length; i++) {
      address storageAttester = _getSPStorage().attestationRegistry[attestationIds[i]].attester;
      if (delegateMode && storageAttester != currentAttester) {
        revert AttestationWrongAttester();
      }
      uint64 schemaId = _revoke(attestationIds[i], reasons[i], delegateMode);
      ISPHook hook = __getResolverFromAttestationId(attestationIds[i]);
      if (address(hook) != address(0)) {
        hook.didReceiveRevocation(
          storageAttester,
          schemaId,
          attestationIds[i],
          resolverFeesERC20Tokens[i],
          resolverFeesERC20Amount[i],
          extraData
        );
      }
    }
    _callGlobalHook();
  }

  function revokeOffchain(
```

```
    string calldata offchainAttestationId,
    string calldata reason,
    bytes calldata delegateSignature
  )
    external
    override
  {
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
      address storageAttester =
_getSPStorage().offchainAttestationRegistry[offchainAttestationId].attester;
      __checkDelegationSignature(
        storageAttester, getDelegatedOffchainRevokeHash(offchainAttestationId, reason),
delegateSignature
      );
    }
    _revokeOffchain(offchainAttestationId, reason, delegateMode);
    _callGlobalHook();
  }

  function revokeOffchainBatch(
    string[] calldata offchainAttestationIds,
    string[] calldata reasons,
    bytes calldata delegateSignature
  )
    external
    override
  {
    address currentAttester = _msgSender();
    bool delegateMode = delegateSignature.length != 0;
    if (delegateMode) {
```

```
        address storageAttester =
_getSPStorage().offchainAttestationRegistry[offchainAttestationIds[0]].attester;
        __checkDelegationSignature(
            storageAttester, getDelegatedOffchainRevokeBatchHash(offchainAttestationIds,
reasons), delegateSignature
        );
        currentAttester = storageAttester;
      }
      for (uint256 i = 0; i < offchainAttestationIds.length; i++) {
        address storageAttester =
_getSPStorage().offchainAttestationRegistry[offchainAttestationIds[i]].attester;
        if (delegateMode && storageAttester != currentAttester) {
          revert AttestationWrongAttester();
        }
        _revokeOffchain(offchainAttestationIds[i], reasons[i], delegateMode);
      }
      _callGlobalHook();
    }

    function getSchema(uint64 schemaId) external view override returns (Schema memory) {
      SPStorage storage $ = _getSPStorage();
      if (schemaId < $.initialSchemaCounter) revert LegacySPRequired();
      return $.schemaRegistry[schemaId];
    }

    function getAttestation(uint64 attestationId) external view override returns (Attestation
memory) {
      SPStorage storage $ = _getSPStorage();
      if (attestationId < $.initialAttestationCounter) revert LegacySPRequired();
      return $.attestationRegistry[attestationId];
    }
```

```
  function getOffchainAttestation(string calldata offchainAttestationId)
    external
    view
    returns (OffchainAttestation memory)
  {
    return _getSPStorage().offchainAttestationRegistry[offchainAttestationId];
  }

  function schemaCounter() external view override returns (uint64) {
    return _getSPStorage().schemaCounter;
  }

  function attestationCounter() external view override returns (uint64) {
    return _getSPStorage().attestationCounter;
  }

  function version() external pure override returns (string memory) {
    return "1.1.3";
  }

  function getDelegatedRegisterHash(Schema memory schema) public pure override returns
(bytes32) {
    return keccak256(abi.encode(REGISTER_ACTION_NAME, schema));
  }

  function getDelegatedAttestHash(Attestation memory attestation) public pure override
returns (bytes32) {
    return keccak256(abi.encode(ATTEST_ACTION_NAME, attestation));
  }

  function getDelegatedAttestBatchHash(Attestation[] memory attestations) public pure
returns (bytes32) {
```

```
        return keccak256(abi.encode(ATTEST_BATCH_ACTION_NAME, attestations));
    }

    function getDelegatedOffchainAttestHash(string memory offchainAttestationId)
        public
        pure
        override
        returns (bytes32)
    {
        return keccak256(abi.encode(ATTEST_OFFCHAIN_ACTION_NAME, offchainAttestationId));
    }

    function getDelegatedOffchainAttestBatchHash(string[] memory offchainAttestationIds)
        public
        pure
        returns (bytes32)
    {
        return keccak256(abi.encode(ATTEST_OFFCHAIN_BATCH_ACTION_NAME,
offchainAttestationIds));
    }

    function getDelegatedRevokeHash(
        uint64 attestationId,
        string memory reason
    )
        public
        pure
        override
        returns (bytes32)
    {
        return keccak256(abi.encode(REVOKE_ACTION_NAME, attestationId, reason));
```

```
  }

  function getDelegatedRevokeBatchHash(
    uint64[] memory attestationIds,
    string[] memory reasons
  )
    public
    pure
    returns (bytes32)
  {
    return keccak256(abi.encode(REVOKE_BATCH_ACTION_NAME, attestationIds, reasons));
  }

  function getDelegatedOffchainRevokeHash(
    string memory offchainAttestationId,
    string memory reason
  )
    public
    pure
    override
    returns (bytes32)
  {
    return keccak256(abi.encode(REVOKE_OFFCHAIN_ACTION_NAME, offchainAttestationId,
reason));
  }

  function getDelegatedOffchainRevokeBatchHash(
    string[] memory offchainAttestationIds,
    string[] memory reasons
  )
    public
```

```
    pure
    returns (bytes32)
  {
    return keccak256(abi.encode(REVOKE_OFFCHAIN_BATCH_ACTION_NAME,
offchainAttestationIds, reasons));
  }

  function _callGlobalHook() internal {
    SPStorage storage $ = _getSPStorage();
    if (address($.globalHook) != address(0)) $.globalHook.callHook(_msgData(), _msgSender());
  }

  function _register(Schema memory schema) internal returns (uint64 schemaId) {
    SPStorage storage $ = _getSPStorage();
    if ($.paused) revert Paused();
    schemaId = $.schemaCounter++;
    schema.timestamp = uint64(block.timestamp);
    $.schemaRegistry[schemaId] = schema;
    emit SchemaRegistered(schemaId);
  }

  // solhint-disable-next-line code-complexity
  function _attest(
    Attestation memory attestation,
    string memory indexingKey,
    bool delegateMode
  )
    internal
    returns (uint64 schemaId, uint64 attestationId)
  {
    SPStorage storage $ = _getSPStorage();
```

```
    if ($.paused) revert Paused();
    attestationId = $.attestationCounter++;
    attestation.attestTimestamp = uint64(block.timestamp);
    // In delegation mode, the attester is already checked ahead of time.
    if (!delegateMode && attestation.attester != _msgSender()) {
      revert AttestationWrongAttester();
    }
    if (attestation.linkedAttestationId > 0 &&
!__attestationExists(attestation.linkedAttestationId)) {
      revert AttestationNonexistent();
    }
    if (
      attestation.linkedAttestationId != 0
        && $.attestationRegistry[attestation.linkedAttestationId].attester != attestation.attester
    ) {
      revert AttestationWrongAttester();
    }
    if (attestation.revoked || attestation.revokeTimestamp > 0) {
      revert AttestationAlreadyRevoked();
    }
    Schema memory s = $.schemaRegistry[attestation.schemaId];
    if (!__schemaExists(attestation.schemaId)) revert SchemaNonexistent();
    if (s.maxValidFor > 0) {
      uint256 attestationValidFor = attestation.validUntil - block.timestamp;
      if (s.maxValidFor < attestationValidFor) {
        revert AttestationInvalidDuration();
      }
    }
    $.attestationRegistry[attestationId] = attestation;
    emit AttestationMade(attestationId, indexingKey);
    return (attestation.schemaId, attestationId);
```

```
  }

  function _attestOffchain(string calldata offchainAttestationId, address attester) internal {
    SPStorage storage $ = _getSPStorage();
    if ($.paused) revert Paused();
    OffchainAttestation storage attestation =
$.offchainAttestationRegistry[offchainAttestationId];
    if (__offchainAttestationExists(offchainAttestationId)) {
      revert OffchainAttestationExists();
    }
    attestation.timestamp = uint64(block.timestamp);
    attestation.attester = attester;
    emit OffchainAttestationMade(offchainAttestationId);
  }

  function _revoke(
    uint64 attestationId,
    string memory reason,
    bool delegateMode
  )
    internal
    returns (uint64 schemaId)
  {
    SPStorage storage $ = _getSPStorage();
    if ($.paused) revert Paused();
    Attestation storage a = $.attestationRegistry[attestationId];
    if (a.attester == address(0)) revert AttestationNonexistent();
    // In delegation mode, the attester is already checked ahead of time.
    if (!delegateMode && a.attester != _msgSender()) revert AttestationWrongAttester();
    Schema memory s = $.schemaRegistry[a.schemaId];
    if (!s.revocable) revert AttestationIrrevocable();
```

```
        if (a.revoked) revert AttestationAlreadyRevoked();
        a.revoked = true;
        a.revokeTimestamp = uint64(block.timestamp);
        emit AttestationRevoked(attestationId, reason);
        return a.schemaId;
    }

    function _revokeOffchain(
        string calldata offchainAttestationId,
        string calldata reason,
        bool delegateMode
    )
        internal
    {
        SPStorage storage $ = _getSPStorage();
        if ($.paused) revert Paused();
        OffchainAttestation storage attestation =
$.offchainAttestationRegistry[offchainAttestationId];
        if (!__offchainAttestationExists(offchainAttestationId)) {
            revert OffchainAttestationNonexistent();
        }
        if (!delegateMode && attestation.attester != _msgSender()) {
            revert AttestationWrongAttester();
        }
        if (attestation.timestamp == 1) {
            revert OffchainAttestationAlreadyRevoked();
        }
        attestation.timestamp = 1;
        emit OffchainAttestationRevoked(offchainAttestationId, reason);
    }
```

```
    // solhint-disable-next-line no-empty-blocks
    function _authorizeUpgrade(address newImplementation) internal virtual override onlyOwner
{ }


    function __checkDelegationSignature(
        address delegateAttester,
        bytes32 hash,
        bytes memory delegateSignature
    )
        internal
        view
    {
        if (
            !SignatureChecker.isValidSignatureNow(
                delegateAttester, MessageHashUtils.toEthSignedMessageHash(hash),
delegateSignature
            )
        ) {
            revert InvalidDelegateSignature();
        }
    }


    function __getResolverFromAttestationId(uint64 attestationId) internal view returns (ISPHook)
{
        SPStorage storage $ = _getSPStorage();
        Attestation memory a = $.attestationRegistry[attestationId];
        Schema memory s = $.schemaRegistry[a.schemaId];
        return s.hook;
    }


    function __schemaExists(uint64 schemaId) internal view returns (bool) {
        return _getSPStorage().schemaRegistry[schemaId].timestamp > 0;
```

```
    }

    function __attestationExists(uint64 attestationId) internal view returns (bool) {
        SPStorage storage $ = _getSPStorage();
        return attestationId < $.attestationCounter;
    }

    function __offchainAttestationExists(string memory attestationId) internal view returns (bool) {
        SPStorage storage $ = _getSPStorage();
        return $.offchainAttestationRegistry[attestationId].timestamp != 0;
    }
}
```

ISP.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import { IVersionable } from "./IVersionable.sol";
import { Schema } from "../models/Schema.sol";
import { Attestation, OffchainAttestation } from "../models/Attestation.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

/**
 * @title Sign Protocol Interface
 * @author Jack Xu @ EthSign
 */
interface ISP is IVersionable {
    event SchemaRegistered(uint64 schemaId);
    event AttestationMade(uint64 attestationId, string indexingKey);
    event AttestationRevoked(uint64 attestationId, string reason);
    event OffchainAttestationMade(string attestationId);
    event OffchainAttestationRevoked(string attestationId, string reason);
```

```
/**
 * @dev 0x9e87fac8
 */
error Paused();
/**
 * @dev 0x38f8c6c4
 */
error SchemaNonexistent();
/**
 * @dev 0x71984561
 */
error SchemaWrongRegistrant();
/**
 * @dev 0x8ac42f49
 */
error AttestationIrrevocable();
/**
 * @dev 0x54681a13
 */
error AttestationNonexistent();
/**
 * @dev 0xa65e02ed
 */
error AttestationInvalidDuration();
/**
 * @dev 0xd8c3da86
 */
error AttestationAlreadyRevoked();
/**
 * @dev 0xa9ad2007
```

```
    */
    error AttestationWrongAttester();
    /**
     * @dev 0xc83e3cdf
     */
    error OffchainAttestationExists();
    /**
     * @dev 0xa006519a
     */
    error OffchainAttestationNonexistent();
    /**
     * @dev 0xa0671d20
     */
    error OffchainAttestationAlreadyRevoked();
    /**
     * @dev 0xfdf4e6f9
     */
    error InvalidDelegateSignature();
    /**
     * @dev 0x5c34b9cc
     */
    error LegacySPRequired();


    /**
     * @notice Registers a Schema.
     * @dev Emits `SchemaRegistered`.
     * @param schema See `Schema`.
     * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated attestation. Use `""` or `0x`
     * otherwise.
     * @return schemaId The assigned ID of the registered schema.
```

```
    */
    function register(Schema memory schema, bytes calldata delegateSignature) external returns (uint64 schemaId);

    /**
     * @notice Makes an attestation.
     * @dev Emits `AttestationMade`.
     * @param attestation See `Attestation`.
     * @param indexingKey Used by the frontend to aid indexing.
     * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated attestation. Use `""` or `0x`
     * otherwise.
     * @param extraData This is forwarded to the resolver directly.
     * @return attestationId The assigned ID of the attestation.
     */
    function attest(
        Attestation calldata attestation,
        string calldata indexingKey,
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external
        returns (uint64 attestationId);

    /**
     * @notice Makes an attestation where the schema hook expects ETH payment.
     * @dev Emits `AttestationMade`.
     * @param attestation See `Attestation`.
     * @param resolverFeesETH Amount of funds to send to the hook.
     * @param indexingKey Used by the frontend to aid indexing.
     * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated attestation. Use `""` or `0x`
```

```
     * otherwise.
     * @param extraData This is forwarded to the resolver directly.
     * @return attestationId The assigned ID of the attestation.
     */
    function attest(
        Attestation calldata attestation,
        uint256 resolverFeesETH,
        string calldata indexingKey,
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external
        payable
        returns (uint64 attestationId);

    /**
     * @notice Makes an attestation where the schema hook expects ERC20 payment.
     * @dev Emits `AttestationMade`.
     * @param attestation See `Attestation`.
     * @param resolverFeesERC20Token ERC20 token address used for payment.
     * @param resolverFeesERC20Amount Amount of funds to send to the hook.
     * @param indexingKey Used by the frontend to aid indexing.
     * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated
attestation. Use `""` or `0x`
     * otherwise.
     * @param extraData This is forwarded to the resolver directly.
     * @return attestationId The assigned ID of the attestation.
     */
    function attest(
        Attestation calldata attestation,
        IERC20 resolverFeesERC20Token,
```

```
    uint256 resolverFeesERC20Amount,
    string calldata indexingKey,
    bytes calldata delegateSignature,
    bytes calldata extraData
  )
    external
    returns (uint64 attestationId);

  /**
   * @notice Timestamps an off-chain data ID.
   * @dev Emits `OffchainAttestationMade`.
   * @param offchainAttestationId The off-chain data ID.
   * @param delegateAttester An optional delegated attester that authorized the caller to attest on their behalf if
   * this is a delegated attestation. Use `address(0)` otherwise.
   * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated attestation. Use `""` or `0x`
   * otherwise. Use `""` or `0x` otherwise.
   */
  function attestOffchain(
    string calldata offchainAttestationId,
    address delegateAttester,
    bytes calldata delegateSignature
  )
    external;

  /**
   * @notice Revokes an existing revocable attestation.
   * @dev Emits `AttestationRevoked`. Must be called by the attester.
   * @param attestationId An existing attestation ID.
   * @param reason The revocation reason. This is only emitted as an event to save gas.
```

```
    * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated revocation.
    * @param extraData This is forwarded to the resolver directly.
    */
   function revoke(
      uint64 attestationId,
      string calldata reason,
      bytes calldata delegateSignature,
      bytes calldata extraData
   )
      external;

   /**
    * @notice Revokes an existing revocable attestation where the schema hook expects ERC20 payment.
    * @dev Emits `AttestationRevoked`. Must be called by the attester.
    * @param attestationId An existing attestation ID.
    * @param reason The revocation reason. This is only emitted as an event to save gas.
    * @param resolverFeesETH Amount of funds to send to the hook.
    * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated revocation.
    * @param extraData This is forwarded to the resolver directly.
    */
   function revoke(
      uint64 attestationId,
      string calldata reason,
      uint256 resolverFeesETH,
      bytes calldata delegateSignature,
      bytes calldata extraData
   )
      external
      payable;
```

```
  /**
   * @notice Revokes an existing revocable attestation where the schema hook expects ERC20 payment.
   * @dev Emits `AttestationRevoked`. Must be called by the attester.
   * @param attestationId An existing attestation ID.
   * @param reason The revocation reason. This is only emitted as an event to save gas.
   * @param resolverFeesERC20Token ERC20 token address used for payment.
   * @param resolverFeesERC20Amount Amount of funds to send to the hook.
   * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated revocation.
   * @param extraData This is forwarded to the resolver directly.
   */
  function revoke(
    uint64 attestationId,
    string calldata reason,
    IERC20 resolverFeesERC20Token,
    uint256 resolverFeesERC20Amount,
    bytes calldata delegateSignature,
    bytes calldata extraData
  )
    external;

  /**
   * @notice Revokes an existing offchain attestation.
   * @dev Emits `OffchainAttestationRevoked`. Must be called by the attester.
   * @param offchainAttestationId An existing attestation ID.
   * @param reason The revocation reason. This is only emitted as an event to save gas.
   * @param delegateSignature An optional ECDSA delegateSignature if this is a delegated revocation.
   */
  function revokeOffchain(
```

```
    string calldata offchainAttestationId,
    string calldata reason,
    bytes calldata delegateSignature
)
    external;

/**
 * @notice Batch attests.
 */
function attestBatch(
    Attestation[] calldata attestations,
    string[] calldata indexingKeys,
    bytes calldata delegateSignature,
    bytes calldata extraData
)
    external
    returns (uint64[] calldata attestationIds);

/**
 * @notice Batch attests where the schema hook expects ETH payment.
 */
function attestBatch(
    Attestation[] calldata attestations,
    uint256[] calldata resolverFeesETH,
    string[] calldata indexingKeys,
    bytes calldata delegateSignature,
    bytes calldata extraData
)
    external
    payable
    returns (uint64[] calldata attestationIds);
```

```
/**
 * @notice Batch attests where the schema hook expects ERC20 payment.
 */
function attestBatch(
    Attestation[] calldata attestations,
    IERC20[] calldata resolverFeesERC20Tokens,
    uint256[] calldata resolverFeesERC20Amount,
    string[] calldata indexingKeys,
    bytes calldata delegateSignature,
    bytes calldata extraData
)
    external
    returns (uint64[] calldata attestationIds);

/**
 * @notice Batch timestamps off-chain data IDs.
 */
function attestOffchainBatch(
    string[] calldata offchainAttestationIds,
    address delegateAttester,
    bytes calldata delegateSignature
)
    external;

/**
 * @notice Batch revokes revocable on-chain attestations.
 */
function revokeBatch(
    uint64[] calldata attestationIds,
    string[] calldata reasons,
```

```
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external;

    /**
     * @notice Batch revokes revocable on-chain attestations where the schema hook expects ETH payment.
     */
    function revokeBatch(
        uint64[] calldata attestationIds,
        string[] calldata reasons,
        uint256[] calldata resolverFeesETH,
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external
        payable;

    /**
     * @notice Batch revokes revocable on-chain attestations where the schema hook expects ERC20 payment.
     */
    function revokeBatch(
        uint64[] calldata attestationIds,
        string[] calldata reasons,
        IERC20[] calldata resolverFeesERC20Tokens,
        uint256[] calldata resolverFeesERC20Amount,
        bytes calldata delegateSignature,
        bytes calldata extraData
    )
        external;
```

```
/**
 * @notice Batch revokes off-chain attestations.
 */
function revokeOffchainBatch(
    string[] calldata offchainAttestationIds,
    string[] calldata reasons,
    bytes calldata delegateSignature
)
    external;

/**
 * @notice Returns the specified `Schema`.
 */
function getSchema(uint64 schemaId) external view returns (Schema calldata);

/**
 * @notice Returns the specified `Attestation`.
 */
function getAttestation(uint64 attestationId) external view returns (Attestation calldata);

/**
 * @notice Returns the specified `OffchainAttestation`.
 */
function getOffchainAttestation(string calldata offchainAttestationId)
    external
    view
    returns (OffchainAttestation calldata);

/**
 * @notice Returns the hash that will be used to authorize a delegated registration.
```

```
    */
    function getDelegatedRegisterHash(Schema memory schema) external pure returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated attestation.
     */
    function getDelegatedAttestHash(Attestation calldata attestation) external pure returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated batch attestation.
     */
    function getDelegatedAttestBatchHash(Attestation[] calldata attestations) external pure returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated offchain attestation.
     */
    function getDelegatedOffchainAttestHash(string calldata offchainAttestationId) external pure returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated batch offchain attestation.
     */
    function getDelegatedOffchainAttestBatchHash(string[] calldata offchainAttestationIds)
        external
        pure
        returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated revocation.
```

```
    */
    function getDelegatedRevokeHash(uint64 attestationId, string memory reason) external pure returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated batch revocation.
     */
    function getDelegatedRevokeBatchHash(
        uint64[] memory attestationIds,
        string[] memory reasons
    )
        external
        pure
        returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated offchain revocation.
     */
    function getDelegatedOffchainRevokeHash(
        string memory offchainAttestationId,
        string memory reason
    )
        external
        pure
        returns (bytes32);

    /**
     * @notice Returns the hash that will be used to authorize a delegated batch offchain revocation.
     */
    function getDelegatedOffchainRevokeBatchHash(
        string[] memory offchainAttestationIds,
```

```
        string[] memory reasons
    )
        external
        pure
        returns (bytes32);

    /**
     * @notice Returns the current schema counter. This is incremented for each `Schema` registered.
     */
    function schemaCounter() external view returns (uint64);

    /**
     * @notice Returns the current on-chain attestation counter. This is incremented for each `Attestation` made.
     */
    function attestationCounter() external view returns (uint64);
}
```

ISPGlobalHook.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

interface ISPGlobalHook {
    function callHook(bytes calldata msgData, address msgSender) external;
}
```

ISPHook.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
/**
 * @title SIGN Attestation Protocol Resolver Interface
 * @author Jack Xu @ EthSign
 */
interface ISPHook {
    function didReceiveAttestation(
        address attester,
        uint64 schemaId,
        uint64 attestationId,
        bytes calldata extraData
    )
        external
        payable;

    function didReceiveAttestation(
        address attester,
        uint64 schemaId,
        uint64 attestationId,
        IERC20 resolverFeeERC20Token,
        uint256 resolverFeeERC20Amount,
        bytes calldata extraData
    )
        external;

    function didReceiveRevocation(
        address attester,
        uint64 schemaId,
        uint64 attestationId,
        bytes calldata extraData
    )
```

```
        external
        payable;

    function didReceiveRevocation(
        address attester,
        uint64 schemaId,
        uint64 attestationId,
        IERC20 resolverFeeERC20Token,
        uint256 resolverFeeERC20Amount,
        bytes calldata extraData
    )
        external;
}
```

IVersionable.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

/**
 * @title IVersionable
 * @author Jack Xu @ EthSign
 * @dev This interface helps contracts to keep track of their versioning for upgrade compatibility checks.
 */
interface IVersionable {
    function version() external pure returns (string memory);
}
```

MockERC20.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
  constructor() ERC20("MockERC20", "M20") { }

  function mint(address to, uint256 amount) public {
    _mint(to, amount);
  }
}
```

MockResolver.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import { ISPHook, IERC20 } from "../interfaces/ISPHook.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

contract MockResolverAdmin is OwnableUpgradeable {
  using SafeERC20 for IERC20;

  mapping(uint64 schemaId => uint256 ethFees) public schemaAttestETHFees;
  mapping(uint64 schemaId => mapping(IERC20 tokenAddress => uint256 tokenFees)) public schemaAttestTokenFees;
  mapping(uint64 attestationId => uint256 ethFees) public attestationETHFees;
  mapping(uint64 attestationId => mapping(IERC20 tokenAddress => uint256 tokenFees)) public attestationTokenFees;
  mapping(IERC20 tokenAddress => bool approved) public approvedTokens;

  event ETHFeesReceived(uint64 attestationId, uint256 amount);
  event TokenFeesReceived(uint64 attestationId, IERC20 token, uint256 amount);
```

```
    error MismatchETHFee();
    error InsufficientETHFee();
    error UnapprovedToken();
    error InsufficientTokenFee();

    function initialize() external initializer {
        __Ownable_init(_msgSender());
    }

    function setSchemaAttestETHFees(uint64 schemaId, uint256 fees) external onlyOwner {
        schemaAttestETHFees[schemaId] = fees;
    }

    function setSchemaAttestTokenFees(uint64 schemaId, IERC20 token, uint256 fees) external
onlyOwner {
        schemaAttestTokenFees[schemaId][token] = fees;
    }

    function setFeeTokenApprovalStatus(IERC20 token, bool approved) external onlyOwner {
        approvedTokens[token] = approved;
    }

    function _receiveEther(address attester, uint64 schemaId, uint64 attestationId) internal {
        uint256 fees =
            schemaAttestETHFees[schemaId] == 0 ? attestationETHFees[attestationId] :
schemaAttestETHFees[schemaId];
        if (msg.value != fees) revert InsufficientETHFee();
        emit ETHFeesReceived(attestationId, msg.value);
        attester;
    }
```

```
  function _receiveTokens(
    address attester,
    uint64 schemaId,
    uint64 attestationId,
    IERC20 resolverFeeERC20Token,
    uint256 resolverFeeERC20Amount
  )
    internal
  {
    if (!approvedTokens[resolverFeeERC20Token]) revert UnapprovedToken();
    uint256 fees = schemaAttestTokenFees[schemaId][resolverFeeERC20Token] == 0
      ? attestationTokenFees[attestationId][resolverFeeERC20Token]
      : schemaAttestTokenFees[schemaId][resolverFeeERC20Token];
    if (resolverFeeERC20Amount != fees) revert InsufficientTokenFee();
    resolverFeeERC20Token.safeTransferFrom(attester, address(this),
resolverFeeERC20Amount);
    emit TokenFeesReceived(attestationId, resolverFeeERC20Token,
resolverFeeERC20Amount);
  }
}

contract MockResolver is ISPHook, MockResolverAdmin {
  function didReceiveAttestation(
    address attester,
    uint64 schemaId,
    uint64 attestationId,
    bytes calldata
  )
    external
    payable
    override
  // solhint-disable-next-line no-empty-blocks
```

```
{}

function didReceiveAttestation(
    address attester,
    uint64 schemaId,
    uint64 attestationId,
    IERC20 resolverFeeERC20Token,
    uint256 resolverFeeERC20Amount,
    bytes calldata
)
    external
    override
{
    _receiveTokens(attester, schemaId, attestationId, resolverFeeERC20Token,
resolverFeeERC20Amount);
}

function didReceiveRevocation(
    address attester,
    uint64 schemaId,
    uint64 attestationId,
    bytes calldata
)
    external
    payable
    override
{
    _receiveEther(attester, schemaId, attestationId);
}

function didReceiveRevocation(
```

```
        address attester,
        uint64 schemaId,
        uint64 attestationId,
        IERC20 resolverFeeERC20Token,
        uint256 resolverFeeERC20Amount,
        bytes calldata
    )
        external
        override
    {
        _receiveTokens(attester, schemaId, attestationId, resolverFeeERC20Token,
resolverFeeERC20Amount);
    }
}
```

Attestation.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import { DataLocation } from "./DataLocation.sol";

/**
 * @title Attestation
 * @author Jack Xu @ EthSign
 * @notice This struct represents an on-chain attestation record. This record is not deleted after
revocation.
 *
 * `schemaId`: The `Schema` that this Attestation is based on. It must exist.
 * `linkedAttestationId`: Useful if the current Attestation references a previous Attestation. It
can either be 0 or an
 * existing attestation ID.
 * `attestTimestamp`: When the attestation was made. This is automatically populated by
`_attest(...)`.
```

* `revokeTimestamp`: When the attestation was revoked. This is automatically populated by `_revoke(...)`.

* `attester`: The attester. At this time, the attester must be the caller of `attest()`.

* `validUntil`: The expiration timestamp of the Attestation. Must respect `Schema.maxValidFor`. 0 indicates no

* expiration date.

* `dataLocation`: Where `Attestation.data` is stored. See `DataLocation.DataLocation`.

* `revoked`: If the Attestation has been revoked. It is possible to make a revoked Attestation.

* `recipients`: The intended ABI-encoded recipients of this Attestation. This is of type `bytes` to support non-EVM

* repicients.

* `data`: The raw data of the Attestation based on `Schema.schema`. There is no enforcement here, however. Recommended

* to use `abi.encode`.

*/

```
struct Attestation {
    uint64 schemaId;
    uint64 linkedAttestationId;
    uint64 attestTimestamp;
    uint64 revokeTimestamp;
    address attester;
    uint64 validUntil;
    DataLocation dataLocation;
    bool revoked;
    bytes[] recipients;
    bytes data;
}
```

```
/**
 * @title OffchainAttestation
 * @author Jack Xu @ EthSign
 * @notice This struct represents an off-chain attestation record. This record is not deleted after revocation.
```

```
 *
 * `attester`: The attester. At this time, the attester must be the caller of `attestOffchain()`.
 * `timestamp`: The `block.timestamp` of the function call.
 */
struct OffchainAttestation {
    address attester;
    uint64 timestamp;
}
```

DataLocation.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

/**
 * @title DataLocation
 * @author Jack Xu @ EthSign
 * @notice This enum indicates where `Schema.data` and `Attestation.data` are stored.
 */
enum DataLocation {
    ONCHAIN,
    ARWEAVE,
    IPFS,
    CUSTOM
}
```

Schema.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import { ISPHook } from "../interfaces/ISPHook.sol";
import { DataLocation } from "./DataLocation.sol";
```

```
/**
 * @title Schema
 * @author Jack Xu @ EthSign
 * @notice This struct represents an on-chain Schema that Attestations can conform to.
 *
 * `registrant`: The address that registered this schema.
 * `revocable`: Whether Attestations that adopt this Schema can be revoked.
 * `dataLocation`: Where `Schema.data` is stored. See `DataLocation.DataLocation`.
 * `maxValidFor`: The maximum number of seconds that an Attestation can remain valid. 0 means Attestations can be valid
 * forever. This is enforced through `Attestation.validUntil`.
 * `hook`: The `ISPHook` that is called at the end of every function. 0 means there is no hook set. See
 * `ISPHook`.
 * `timestamp`: When the schema was registered. This is automatically populated by `_register(...)`.
 * `data`: The raw schema that `Attestation.data` should follow. Since there is no way to enforce this, it is a `string`
 * for easy readability.
 */
struct Schema {
    address registrant;
    bool revocable;
    DataLocation dataLocation;
    uint64 maxValidFor;
    ISPHook hook;
    uint64 timestamp;
    string data;
}
```