LICENSE
=======

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and

subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

Cargo.lock
==========

This file is automatically @generated by Cargo.

```
# It is not intended for manual editing.
version = 3

[[package]]
name = "addr2line"
version = "0.21.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8a30b2e23b9e17a9f90641c7ab1549cd9b44f296d3ccbf309d2863cfe398a0cb"
dependencies = [
 "gimli",
]

[[package]]
name = "adler"
version = "1.0.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f26201604c87b1e01bd3d98f8d5d9a8fcbb815e8cedb41ffccbeb4bf593a35fe"

[[package]]
name = "alloy-consensus"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996
dependencies = [
 "alloy-eips",
 "alloy-primitives",
 "alloy-rlp",
 "alloy-serde",
 "c-kzg",
 "serde",
 "sha2",
]

[[package]]
name = "alloy-eips"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996
dependencies = [
 "alloy-primitives",
 "alloy-rlp",
 "alloy-serde",
 "c-kzg",
 "once_cell",
 "serde",
]

[[package]]
name = "alloy-genesis"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996
dependencies = [
 "alloy-primitives",
 "alloy-serde",
 "serde",
]

[[package]]
name = "alloy-json-rpc"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996
dependencies = [
 "alloy-primitives",
 "serde",
 "serde_json",
 "thiserror",
 "tracing",
]
```

```
[[package]]
name = "alloy-primitives"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "99bbad0a6b588ef4aec1b5ddbbfdacd9ef04e00b979617765b03174318ee1f3a"
dependencies = [
 "alloy-rlp",
 "bytes",
 "cfg-if",
 "const-hex",
 "derive_more",
 "hex-literal",
 "itoa",
 "k256",
 "keccak-asm",
 "proptest",
 "rand",
 "ruint",
 "serde",
 "tiny-keccak",
]

[[package]]
name = "alloy-rlp"
version = "0.3.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8d58d9f5da7b40e9bfff0b7e7816700be4019db97d4b6359fe7f94a9e22e42ac"
dependencies = [
 "alloy-rlp-derive",
 "arrayvec",
 "bytes",
]

[[package]]
name = "alloy-rlp-derive"
version = "0.3.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1a047897373be4bbb0224c1afdabca92648dc57a9c9ef6e7b0be3aff7a859c83"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "alloy-rpc-types"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996b
dependencies = [
 "alloy-consensus",
 "alloy-eips",
 "alloy-genesis",
 "alloy-primitives",
 "alloy-rlp",
 "alloy-serde",
 "alloy-sol-types",
 "itertools 0.12.1",
 "serde",
 "serde_json",
 "thiserror",
]

[[package]]
name = "alloy-serde"
version = "0.1.0"
```

```
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996b
dependencies = [
 "alloy-primitives",
 "serde",
 "serde_json",
]

[[package]]
name = "alloy-sol-macro"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "452d929748ac948a10481fff4123affead32c553cf362841c5103dd508bdfc16"
dependencies = [
 "alloy-sol-macro-input",
 "const-hex",
 "heck 0.4.1",
 "indexmap",
 "proc-macro-error",
 "proc-macro2",
 "quote",
 "syn 2.0.48",
 "syn-solidity",
 "tiny-keccak",
]

[[package]]
name = "alloy-sol-macro-input"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "df64e094f6d2099339f9e82b5b38440b159757b6920878f28316243f8166c8d1"
dependencies = [
 "const-hex",
 "dunce",
 "heck 0.5.0",
 "proc-macro2",
 "quote",
 "syn 2.0.48",
 "syn-solidity",
]

[[package]]
name = "alloy-sol-types"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "43bc2d6dfc2a19fd56644494479510f98b1ee929e04cf0d4aa45e98baa3e545b"
dependencies = [
 "alloy-primitives",
 "alloy-sol-macro",
 "const-hex",
]

[[package]]
name = "alloy-transport"
version = "0.1.0"
source = "git+https://github.com/alloy-rs/alloy.git?rev=cad7935#cad7935d69f433e45d190902e58b1c996b
dependencies = [
 "alloy-json-rpc",
 "base64",
 "futures-util",
 "futures-utils-wasm",
 "serde",
 "serde_json",
 "thiserror",
 "tokio",
 "tower",
 "url",
```

```
 "wasm-bindgen-futures",
]

[[package]]
name = "anyhow"
version = "1.0.79"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "080e9890a082662b09c1ad45f567faeeb47f22b5fb23895fbe1e651e718e25ca"

[[package]]
name = "ark-ff"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6b3235cc41ee7a12aaaf2c575a2ad7b46713a8a50bda2fc3b003a04845c05dd6"
dependencies = [
 "ark-ff-asm 0.3.0",
 "ark-ff-macros 0.3.0",
 "ark-serialize 0.3.0",
 "ark-std 0.3.0",
 "derivative",
 "num-bigint",
 "num-traits",
 "paste",
 "rustc_version 0.3.3",
 "zeroize",
]

[[package]]
name = "ark-ff"
version = "0.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ec847af850f44ad29048935519032c33da8aa03340876d351dfab5660d2966ba"
dependencies = [
 "ark-ff-asm 0.4.2",
 "ark-ff-macros 0.4.2",
 "ark-serialize 0.4.2",
 "ark-std 0.4.0",
 "derivative",
 "digest 0.10.7",
 "itertools 0.10.5",
 "num-bigint",
 "num-traits",
 "paste",
 "rustc_version 0.4.0",
 "zeroize",
]

[[package]]
name = "ark-ff-asm"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "db02d390bf6643fb404d3d22d31aee1c4bc4459600aef9113833d17e786c6e44"
dependencies = [
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "ark-ff-asm"
version = "0.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3ed4aa4fe255d0bc6d79373f7e31d2ea147bcf486cba1be5ba7ea85abdb92348"
dependencies = [
 "quote",
 "syn 1.0.109",
]
```

```
[[package]]
name = "ark-ff-macros"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "db2fd794a08ccb318058009eefdf15bcaaaaf6f8161eb3345f907222bac38b20"
dependencies = [
 "num-bigint",
 "num-traits",
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "ark-ff-macros"
version = "0.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7abe79b0e4288889c4574159ab790824d0033b9fdcb2a112a3182fac2e514565"
dependencies = [
 "num-bigint",
 "num-traits",
 "proc-macro2",
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "ark-serialize"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1d6c2b318ee6e10f8c2853e73a83adc0ccb88995aa978d8a3408d492ab2ee671"
dependencies = [
 "ark-std 0.3.0",
 "digest 0.9.0",
]

[[package]]
name = "ark-serialize"
version = "0.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "adb7b85a02b83d2f22f89bd5cac66c9c89474240cb6207cb1efc16d098e822a5"
dependencies = [
 "ark-std 0.4.0",
 "digest 0.10.7",
 "num-bigint",
]

[[package]]
name = "ark-std"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1df2c09229cbc5a028b1d70e00fdb2acee28b1055dfb5ca73eea49c5a25c4e7c"
dependencies = [
 "num-traits",
 "rand",
]

[[package]]
name = "ark-std"
version = "0.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "94893f1e0c6eeab764ade8dc4c0db24caf4fe7cbbaafc0eba0a9030f447b5185"
dependencies = [
 "num-traits",
 "rand",
]
```

```
[[package]]
name = "arrayvec"
version = "0.7.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "96d30a06541fbafbc7f82ed10c06164cfbd2c401138f6addd8404629c4b16711"

[[package]]
name = "auto_impl"
version = "1.1.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "823b8bb275161044e2ac7a25879cb3e2480cb403e3943022c7c769c599b756aa"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "autocfg"
version = "1.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d468802bab17cbc0cc575e9b053f41e72aa36bfa6b7f55e3529ffa43161b97fa"

[[package]]
name = "backtrace"
version = "0.3.69"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "2089b7e3f35b9dd2d0ed921ead4f6d318c27680d4a5bd167b3ee120edb105837"
dependencies = [
 "addr2line",
 "cc",
 "cfg-if",
 "libc",
 "miniz_oxide",
 "object",
 "rustc-demangle",
]

[[package]]
name = "base16ct"
version = "0.2.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4c7f02d4ea65f2c1853089ffd8d2787bdbc63de2f0d29dedbcf8ccdfa0ccd4cf"

[[package]]
name = "base64"
version = "0.22.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9475866fec1451be56a3c2400fd081ff546538961565ccb5b7142cbd22bc7a51"

[[package]]
name = "base64ct"
version = "1.6.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8c3c1a368f70d6cf7302d78f8f7093da241fb8e8807c05cc9e51a125895a6d5b"

[[package]]
name = "bincode"
version = "1.3.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b1f45e9417d87227c7a56d22e471c6206462cba514c7590c09aff4cf6d1ddcad"
dependencies = [
 "serde",
]
```

```
[[package]]
name = "bit-set"
version = "0.5.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0700ddab506f33b20a03b13996eccd309a48e5ff77d0d95926aa0210fb4e95f1"
dependencies = [
 "bit-vec",
]

[[package]]
name = "bit-vec"
version = "0.6.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "349f9b6a179ed607305526ca489b34ad0a41aed5f7980fa90eb03160b69598fb"

[[package]]
name = "bitflags"
version = "2.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ed570934406eb16438a4e976b1b4500774099c13b8cb96eec99f620f05090ddf"

[[package]]
name = "bitvec"
version = "1.0.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1bc2832c24239b0141d5674bb9174f9d68a8b5b3f2753311927c172ca46f7e9c"
dependencies = [
 "funty",
 "radium",
 "tap",
 "wyz",
]

[[package]]
name = "block-buffer"
version = "0.10.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3078c7629b62d3f0439517fa394996acacc5cbc91c5a20d8c658e77abd503a71"
dependencies = [
 "generic-array",
]

[[package]]
name = "blst"
version = "0.3.11"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c94087b935a822949d3291a9989ad2b2051ea141eda0fd4e478a75f6aa3e604b"
dependencies = [
 "cc",
 "glob",
 "threadpool",
 "zeroize",
]

[[package]]
name = "bumpalo"
version = "3.14.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7f30e7476521f6f8af1a1c4c0b8cc94f0bee37d91763d0ca2665f299b6cd8aec"

[[package]]
name = "byte-slice-cast"
version = "1.2.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c3ac9f8b63eca6fd385229b3675f6cc0dc5c8a5c8a54a59d4f52ffd670d87b0c"
```

```
[[package]]
name = "byteorder"
version = "1.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1fd0f2584146f6f2ef48085050886acf353beff7305ebd1ae69500e27c67f64b"

[[package]]
name = "bytes"
version = "1.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a2bd12c1caf447e69cd4528f47f94d203fd2582878ecb9e9465484c4148a8223"
dependencies = [
 "serde",
]

[[package]]
name = "c-kzg"
version = "1.0.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3130f3d8717cc02e668a896af24984d5d5d4e8bf12e278e982e0f1bd88a0f9af"
dependencies = [
 "blst",
 "cc",
 "glob",
 "hex",
 "libc",
 "serde",
]

[[package]]
name = "cc"
version = "1.0.83"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f1174fb0b6ec23863f8b971027804a42614e347eafb0a95bf0b12cdae21fc4d0"
dependencies = [
 "libc",
]

[[package]]
name = "cfg-if"
version = "1.0.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "baf1de4339761588bc0619e3cbc0120ee582ebb74b53b4efbf79117bd2da40fd"

[[package]]
name = "const-hex"
version = "1.10.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a5104de16b218eddf8e34ffe2f86f74bfa4e61e95a1b89732fccf6325efd0557"
dependencies = [
 "cfg-if",
 "cpufeatures",
 "hex",
 "proptest",
 "serde",
]

[[package]]
name = "const-oid"
version = "0.9.6"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c2459377285ad874054d797f3ccebf984978aa39129f6eafde5cdc8315b612f8"

[[package]]
name = "convert_case"
version = "0.4.0"
```

```
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6245d59a3e82a7fc217c5828a6692dbc6dfb63a0c8c90495621f7b9d79704a0e"

[[package]]
name = "cpufeatures"
version = "0.2.12"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "53fe5e26ff1b7aef8bca9c6080520cfb8d9333c7568e1829cef191a9723e5504"
dependencies = [
 "libc",
]

[[package]]
name = "crunchy"
version = "0.2.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7a81dae078cea95a014a339291cec439d2f232ebe854a9d672b796c6afafa9b7"

[[package]]
name = "crypto-bigint"
version = "0.5.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0dc92fb57ca44df6db8059111ab3af99a63d5d0f8375d9972e319a379c6bab76"
dependencies = [
 "generic-array",
 "rand_core",
 "subtle",
 "zeroize",
]

[[package]]
name = "crypto-common"
version = "0.1.6"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1bfb12502f3fc46cca1bb51ac28df9d618d813cdc3d2f25b9fe775a34af26bb3"
dependencies = [
 "generic-array",
 "typenum",
]

[[package]]
name = "der"
version = "0.7.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "fffa369a668c8af7dbf8b5e56c9f744fbd399949ed171606040001947de40b1c"
dependencies = [
 "const-oid",
 "zeroize",
]

[[package]]
name = "derivative"
version = "2.2.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "fcc3dd5e9e9c0b295d6e1e4d811fb6f157d5ffd784b8d202fc62eac8035a770b"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "derive_more"
version = "0.99.17"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4fb810d30a7c1953f91334de7244731fc3f3c10d7fe163338a35b9f640960321"
```

```
dependencies = [
 "convert_case",
 "proc-macro2",
 "quote",
 "rustc_version 0.4.0",
 "syn 1.0.109",
]

[[package]]
name = "digest"
version = "0.9.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d3dd60d1080a57a05ab032377049e0591415d2b31afd7028356dbf3cc6dcb066"
dependencies = [
 "generic-array",
]

[[package]]
name = "digest"
version = "0.10.7"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9ed9a281f7bc9b7576e61468ba615a66a5c8cfdff42420a70aa82701a3b1e292"
dependencies = [
 "block-buffer",
 "const-oid",
 "crypto-common",
 "subtle",
]

[[package]]
name = "dunce"
version = "1.0.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "56ce8c6da7551ec6c462cbaf3bfbc75131ebbfa1c944aeaa9dab51ca1c5f0c3b"

[[package]]
name = "ecdsa"
version = "0.16.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ee27f32b5c5292967d2d4a9d7f1e0b0aed2c15daded5a60300e4abb9d8020bca"
dependencies = [
 "der",
 "digest 0.10.7",
 "elliptic-curve",
 "rfc6979",
 "signature",
 "spki",
]

[[package]]
name = "either"
version = "1.9.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a26ae43d7bcc3b814de94796a5e736d4029efb0ee900c12e2d54c993ad1a1e07"

[[package]]
name = "elliptic-curve"
version = "0.13.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b5e6043086bf7973472e0c7dff2142ea0b680d30e18d9cc40f267efbf222bd47"
dependencies = [
 "base16ct",
 "crypto-bigint",
 "digest 0.10.7",
 "ff",
 "generic-array",
```

```
 "group",
 "pkcs8",
 "rand_core",
 "sec1",
 "subtle",
 "zeroize",
]

[[package]]
name = "equivalent"
version = "1.0.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "5443807d6dff69373d433ab9ef5378ad8df50ca6298caf15de6e52e24aaf54d5"

[[package]]
name = "errno"
version = "0.3.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a258e46cdc063eb8519c00b9fc845fc47bcfca4130e2f08e88665ceda8474245"
dependencies = [
 "libc",
 "windows-sys",
]

[[package]]
name = "fastrand"
version = "2.0.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "25cbce373ec4653f1a01a31e8a5e5ec0c622dc27ff9c4e6606eefef5cbbed4a5"

[[package]]
name = "fastrlp"
version = "0.3.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "139834ddba373bbdd213dffe02c8d110508dcf1726c2be27e8d1f7d7e1856418"
dependencies = [
 "arrayvec",
 "auto_impl",
 "bytes",
]

[[package]]
name = "ff"
version = "0.13.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ded41244b729663b1e574f1b4fb731469f69f79c17667b5d776b16cda0479449"
dependencies = [
 "rand_core",
 "subtle",
]

[[package]]
name = "fixed-hash"
version = "0.8.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "835c052cb0c08c1acf6ffd71c022172e18723949c8282f2b9f27efbc51e64534"
dependencies = [
 "byteorder",
 "rand",
 "rustc-hex",
 "static_assertions",
]

[[package]]
name = "fnv"
version = "1.0.7"
```

```
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3f9eec918d3f24069decb9af1554cad7c880e2da24a9afd88aca000531ab82c1"

[[package]]
name = "form_urlencoded"
version = "1.2.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e13624c2627564efccf4934284bdd98cbaa14e79b0b5a141218e507b3a823456"
dependencies = [
 "percent-encoding",
]

[[package]]
name = "funty"
version = "2.0.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e6d5a32815ae3f33302d95fdcb2ce17862f8c65363dcfd29360480ba1001fc9c"

[[package]]
name = "futures-core"
version = "0.3.30"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "dfc6580bb841c5a68e9ef15c77ccc837b40a7504914d52e47b8b0e9bbda25a1d"

[[package]]
name = "futures-macro"
version = "0.3.30"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "87750cf4b7a4c0625b1529e4c543c2182106e4dedc60a2a6455e00d212c489ac"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "futures-task"
version = "0.3.30"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "38d84fa142264698cdce1a9f9172cf383a0c82de1bddcf3092901442c4097004"

[[package]]
name = "futures-util"
version = "0.3.30"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3d6401deb83407ab3da39eba7e33987a73c3df0c82b4bb5813ee871c19c41d48"
dependencies = [
 "futures-core",
 "futures-macro",
 "futures-task",
 "pin-project-lite",
 "pin-utils",
 "slab",
]

[[package]]
name = "futures-utils-wasm"
version = "0.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "42012b0f064e01aa58b545fe3727f90f7dd4020f4a3ea735b50344965f5a57e9"

[[package]]
name = "generic-array"
version = "0.14.7"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "85649ca51fd72272d7821adaf274ad91c288277713d9c18820d8499a7ff69e9a"
```

```
dependencies = [
 "typenum",
 "version_check",
 "zeroize",
]

[[package]]
name = "getrandom"
version = "0.2.12"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "190092ea657667030ac6a35e305e62fc4dd69fd98ac98631e5d3a2b1575a12b5"
dependencies = [
 "cfg-if",
 "libc",
 "wasi",
]

[[package]]
name = "gimli"
version = "0.28.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4271d37baee1b8c7e4b708028c57d816cf9d2434acb33a549475f78c181f6253"

[[package]]
name = "glob"
version = "0.3.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d2fabcfbdc87f4758337ca535fb41a6d701b65693ce38287d856d1674551ec9b"

[[package]]
name = "group"
version = "0.13.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f0f9ef7462f7c099f518d754361858f86d8a07af53ba9af0fe635bbccb151a63"
dependencies = [
 "ff",
 "rand_core",
 "subtle",
]

[[package]]
name = "hashbrown"
version = "0.14.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "290f1a1d9242c78d09ce40a5e87e7554ee637af1351968159f4952f028f75604"

[[package]]
name = "heck"
version = "0.4.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "95505c38b4572b2d910cecb0281560f54b440a19336cbbcb27bf6ce6adc6f5a8"
dependencies = [
 "unicode-segmentation",
]

[[package]]
name = "heck"
version = "0.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "2304e00983f87ffb38b55b444b5e3b60a884b5d30c0fca7d82fe33449bbe55ea"

[[package]]
name = "hermit-abi"
version = "0.3.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d231dfb89cfffdbc30e7fc41579ed6066ad03abda9e567ccafae602b97ec5024"
```

```
[[package]]
name = "hex"
version = "0.4.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7f24254aa9a54b5c858eaee2f5bccdb46aaf0e486a595ed5fd8f86ba55232a70"
dependencies = [
 "serde",
]

[[package]]
name = "hex-literal"
version = "0.4.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6fe2267d4ed49bc07b63801559be28c718ea06c4738b7a03c94df7386d2cde46"

[[package]]
name = "hmac"
version = "0.12.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6c49c37c09c17a53d937dfbb742eb3a961d65a994e6bcdcf37e7399d0cc8ab5e"
dependencies = [
 "digest 0.10.7",
]

[[package]]
name = "http"
version = "1.0.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b32afd38673a8016f7c9ae69e5af41a58f81b1d31689040f2f1959594ce194ea"
dependencies = [
 "bytes",
 "fnv",
 "itoa",
]

[[package]]
name = "id-arena"
version = "2.2.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "25a2bc672d1148e28034f176e01fffebb08b35768468cc954630da77a1449005"

[[package]]
name = "idna"
version = "0.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "634d9b1461af396cad843f47fdba5597a4f9e6ddd4bfb6ff5d85028c25cb12f6"
dependencies = [
 "unicode-bidi",
 "unicode-normalization",
]

[[package]]
name = "impl-codec"
version = "0.6.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ba6a270039626615617f3f36d15fc827041df3b78c439da2cadfa47455a77f2f"
dependencies = [
 "parity-scale-codec",
]

[[package]]
name = "impl-trait-for-tuples"
version = "0.2.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "11d7a9f6330b71fea57921c9b61c47ee6e84f72d394754eff6163ae67e7395eb"
```

```
dependencies = [
 "proc-macro2",
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "indexmap"
version = "2.2.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "824b2ae422412366ba479e8111fd301f7b5faece8149317bb81925979a53f520"
dependencies = [
 "equivalent",
 "hashbrown",
 "serde",
]

[[package]]
name = "itertools"
version = "0.10.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b0fd2260e829bddf4cb6ea802289de2f86d6a7a690192fbe91b3f46e0f2c8473"
dependencies = [
 "either",
]

[[package]]
name = "itertools"
version = "0.12.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ba291022dbbd398a455acf126c1e341954079855bc60dfdda641363bd6922569"
dependencies = [
 "either",
]

[[package]]
name = "itoa"
version = "1.0.10"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b1a46d1a171d865aa5f83f92695765caa047a9b4cbae2cbf37dbd613a793fd4c"

[[package]]
name = "js-sys"
version = "0.3.67"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9a1d36f1235bc969acba30b7f5990b864423a6068a10f7c90ae8f0112e3a59d1"
dependencies = [
 "wasm-bindgen",
]

[[package]]
name = "k256"
version = "0.13.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "956ff9b67e26e1a6a866cb758f12c6f8746208489e3e4a4b5580802f2f0a587b"
dependencies = [
 "cfg-if",
 "ecdsa",
 "elliptic-curve",
 "once_cell",
 "sha2",
]

[[package]]
name = "keccak-asm"
version = "0.1.0"
```

```
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bb8515fff80ed850aea4a1595f2e519c003e2a00a82fe168ebf5269196caf444"
dependencies = [
 "digest 0.10.7",
 "sha3-asm",
]

[[package]]
name = "kinode_process_lib"
version = "0.7.0"
dependencies = [
 "alloy-json-rpc",
 "alloy-primitives",
 "alloy-rpc-types",
 "alloy-transport",
 "anyhow",
 "bincode",
 "http",
 "mime_guess",
 "rand",
 "rmp-serde",
 "serde",
 "serde_json",
 "thiserror",
 "url",
 "wit-bindgen",
]

[[package]]
name = "lazy_static"
version = "1.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e2abad23fbc42b3700f2f279844dc832adb2b2eb069b2df918f455c4e18cc646"

[[package]]
name = "leb128"
version = "0.2.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "884e2677b40cc8c339eaefcb701c32ef1fd2493d71118dc0ca4b6a736c93bd67"

[[package]]
name = "libc"
version = "0.2.153"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9c198f91728a82281a64e1f4f9eeb25d82cb32a5de251c6bd1b5154d63a8e7bd"

[[package]]
name = "libm"
version = "0.2.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4ec2a862134d2a7d32d7983ddcdd1c4923530833c9f2ea1a44fc5fa473989058"

[[package]]
name = "linux-raw-sys"
version = "0.4.13"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "01cda141df6706de531b6c46c3a33ecca755538219bd484262fa09410c13539c"

[[package]]
name = "log"
version = "0.4.20"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b5e6163cb8c49088c2c36f57875e58ccd8c87c7427f7fbd50ea6710b2f3f2e8f"

[[package]]
name = "memchr"
```

```
version = "2.7.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "523dc4f511e55ab87b694dc30d0f820d60906ef06413f93d4d7a1385599cc149"

[[package]]
name = "mime"
version = "0.3.17"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6877bb514081ee2a7ff5ef9de3281f14a4dd4bceac4c09388074a6b5df8a139a"

[[package]]
name = "mime_guess"
version = "2.0.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4192263c238a5f0d0c6bfd21f336a313a4ce1c450542449ca191bb657b4642ef"
dependencies = [
 "mime",
 "unicase",
]

[[package]]
name = "miniz_oxide"
version = "0.7.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9d811f3e15f28568be3407c8e7fdb6514c1cda3cb30683f15b6a1a1dc4ea14a7"
dependencies = [
 "adler",
]

[[package]]
name = "num-bigint"
version = "0.4.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "608e7659b5c3d7cba262d894801b9ec9d00de989e8a82bd4bef91d08da45cdc0"
dependencies = [
 "autocfg",
 "num-integer",
 "num-traits",
]

[[package]]
name = "num-integer"
version = "0.1.45"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "225d3389fb3509a24c93f5c29eb6bde2586b98d9f016636dff58d7c6f7569cd9"
dependencies = [
 "autocfg",
 "num-traits",
]

[[package]]
name = "num-traits"
version = "0.2.17"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "39e3200413f237f41ab11ad6d161bc7239c84dcb631773ccd7de3dfe4b5c267c"
dependencies = [
 "autocfg",
 "libm",
]

[[package]]
name = "num_cpus"
version = "1.16.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4161fcb6d602d4d2081af7c3a45852d875a03dd337a6bfdd6e06407b61342a43"
dependencies = [
```

```
 "hermit-abi",
 "libc",
]

[[package]]
name = "object"
version = "0.32.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a6a622008b6e321afc04970976f62ee297fdbaa6f95318ca343e3eebb9648441"
dependencies = [
 "memchr",
]

[[package]]
name = "once_cell"
version = "1.19.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3fdb12b2476b595f9358c5161aa467c2438859caa136dec86c26fdd2efe17b92"

[[package]]
name = "parity-scale-codec"
version = "3.6.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "881331e34fa842a2fb61cc2db9643a8fedc615e47cfcc52597d1af0db9a7e8fe"
dependencies = [
 "arrayvec",
 "bitvec",
 "byte-slice-cast",
 "impl-trait-for-tuples",
 "parity-scale-codec-derive",
 "serde",
]

[[package]]
name = "parity-scale-codec-derive"
version = "3.6.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "be30eaf4b0a9fba5336683b38de57bb86d179a35862ba6bfcf57625d006bde5b"
dependencies = [
 "proc-macro-crate",
 "proc-macro2",
 "quote",
 "syn 1.0.109",
]

[[package]]
name = "paste"
version = "1.0.14"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "de3145af08024dea9fa9914f381a17b8fc6034dfb00f3a84013f7ff43f29ed4c"

[[package]]
name = "percent-encoding"
version = "2.3.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e3148f5046208a5d56bcfc03053e3ca6334e51da8dfb19b6cdc8b306fae3283e"

[[package]]
name = "pest"
version = "2.7.7"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "219c0dcc30b6a27553f9cc242972b67f75b60eb0db71f0b5462f38b058c41546"
dependencies = [
 "memchr",
 "thiserror",
 "ucd-trie",
```

```
]

[[package]]
name = "pin-project"
version = "1.1.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0302c4a0442c456bd56f841aee5c3bfd17967563f6fadc9ceb9f9c23cf3807e0"
dependencies = [
 "pin-project-internal",
]

[[package]]
name = "pin-project-internal"
version = "1.1.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "266c042b60c9c76b8d53061e52b2e0d1116abc57cefc8c5cd671619a56ac3690"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "pin-project-lite"
version = "0.2.13"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8afb450f006bf6385ca15ef45d71d2288452bc3683ce2e2cacc0d18e4be60b58"

[[package]]
name = "pin-utils"
version = "0.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8b870d8c151b6f2fb93e84a13146138f05d02ed11c7e7c54f8826aaaf7c9f184"

[[package]]
name = "pkcs8"
version = "0.10.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f950b2377845cebe5cf8b5165cb3cc1a5e0fa5cfa3e1f7f55707d8fd82e0a7b7"
dependencies = [
 "der",
 "spki",
]

[[package]]
name = "ppv-lite86"
version = "0.2.17"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "5b40af805b3121feab8a3c29f04d8ad262fa8e0561883e7653e024ae4479e6de"

[[package]]
name = "primitive-types"
version = "0.12.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0b34d9fd68ae0b74a41b21c03c2f62847aa0ffea044eee893b4c140b37e244e2"
dependencies = [
 "fixed-hash",
 "impl-codec",
 "uint",
]

[[package]]
name = "proc-macro-crate"
version = "2.0.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b00f26d3400549137f92511a46ac1cd8ce37cb5598a96d382381458b992a5d24"
```

```
dependencies = [
 "toml_datetime",
 "toml_edit",
]

[[package]]
name = "proc-macro-error"
version = "1.0.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "da25490ff9892aab3fcf7c36f08cfb902dd3e71ca0f9f9517bea02a73a5ce38c"
dependencies = [
 "proc-macro-error-attr",
 "proc-macro2",
 "quote",
 "syn 1.0.109",
 "version_check",
]

[[package]]
name = "proc-macro-error-attr"
version = "1.0.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a1be40180e52ecc98ad80b184934baf3d0d29f979574e439af5a55274b35f869"
dependencies = [
 "proc-macro2",
 "quote",
 "version_check",
]

[[package]]
name = "proc-macro2"
version = "1.0.78"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e2422ad645d89c99f8f3e6b88a9fdeca7fabeac836b1002371c4367c8f984aae"
dependencies = [
 "unicode-ident",
]

[[package]]
name = "proptest"
version = "1.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "31b476131c3c86cb68032fdc5cb6d5a1045e3e42d96b69fa599fd77701e1f5bf"
dependencies = [
 "bit-set",
 "bit-vec",
 "bitflags",
 "lazy_static",
 "num-traits",
 "rand",
 "rand_chacha",
 "rand_xorshift",
 "regex-syntax",
 "rusty-fork",
 "tempfile",
 "unarray",
]

[[package]]
name = "quick-error"
version = "1.2.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a1d01941d82fa2ab50be1e79e6714289dd7cde78eba4c074bc5a4374f650dfe0"

[[package]]
name = "quote"
```

```
version = "1.0.35"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "291ec9ab5efd934aaf503a6466c5d5251535d108ee747472c3977cc5acc868ef"
dependencies = [
 "proc-macro2",
]

[[package]]
name = "radium"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "dc33ff2d4973d518d823d61aa239014831e521c75da58e3df4840d3f47749d09"

[[package]]
name = "rand"
version = "0.8.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "34af8d1a0e25924bc5b7c43c079c942339d8f0a8b57c39049bef581b46327404"
dependencies = [
 "libc",
 "rand_chacha",
 "rand_core",
]

[[package]]
name = "rand_chacha"
version = "0.3.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e6c10a63a0fa32252be49d21e7709d4d4baf8d231c2dbce1eaa8141b9b127d88"
dependencies = [
 "ppv-lite86",
 "rand_core",
]

[[package]]
name = "rand_core"
version = "0.6.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ec0be4795e2f6a28069bec0b5ff3e2ac9bafc99e6a9a7dc3547996c5c816922c"
dependencies = [
 "getrandom",
]

[[package]]
name = "rand_xorshift"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d25bf25ec5ae4a3f1b92f929810509a2f53d7dca2f50b794ff57e3face536c8f"
dependencies = [
 "rand_core",
]

[[package]]
name = "regex-syntax"
version = "0.8.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c08c74e62047bb2de4ff487b251e4a92e24f48745648451635cec7d591162d9f"

[[package]]
name = "rfc6979"
version = "0.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f8dd2a808d456c4a54e300a23e9f5a67e122c3024119acbfd73e3bf664491cb2"
dependencies = [
 "hmac",
 "subtle",
```

```
]

[[package]]
name = "rlp"
version = "0.5.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bb919243f34364b6bd2fc10ef797edbfa75f33c252e7998527479c6d6b47e1ec"
dependencies = [
 "bytes",
 "rustc-hex",
]

[[package]]
name = "rmp"
version = "0.8.12"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7f9860a6cc38ed1da53456442089b4dfa35e7cedaa326df63017af88385e6b20"
dependencies = [
 "byteorder",
 "num-traits",
 "paste",
]

[[package]]
name = "rmp-serde"
version = "1.1.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bffea85eea980d8a74453e5d02a8d93028f3c34725de143085a844ebe953258a"
dependencies = [
 "byteorder",
 "rmp",
 "serde",
]

[[package]]
name = "ruint"
version = "1.11.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "608a5726529f2f0ef81b8fde9873c4bb829d6b5b5ca6be4d97345ddf0749c825"
dependencies = [
 "alloy-rlp",
 "ark-ff 0.3.0",
 "ark-ff 0.4.2",
 "bytes",
 "fastrlp",
 "num-bigint",
 "num-traits",
 "parity-scale-codec",
 "primitive-types",
 "proptest",
 "rand",
 "rlp",
 "ruint-macro",
 "serde",
 "valuable",
 "zeroize",
]

[[package]]
name = "ruint-macro"
version = "1.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e666a5496a0b2186dbcd0ff6106e29e093c15591bde62c20d3842007c6978a09"

[[package]]
name = "rustc-demangle"
```

```
version = "0.1.23"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d626bb9dae77e28219937af045c257c28bfd3f69333c512553507f5f9798cb76"

[[package]]
name = "rustc-hex"
version = "2.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3e75f6a532d0fd9f7f13144f392b6ad56a32696bfcd9c78f797f16bbb6f072d6"

[[package]]
name = "rustc_version"
version = "0.3.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f0dfe2087c51c460008730de8b57e6a320782fbfb312e1f4d520e6c6fae155ee"
dependencies = [
 "semver 0.11.0",
]

[[package]]
name = "rustc_version"
version = "0.4.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bfa0f585226d2e68097d4f95d113b15b83a82e819ab25717ec0590d9584ef366"
dependencies = [
 "semver 1.0.21",
]

[[package]]
name = "rustix"
version = "0.38.31"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "6ea3e1a662af26cd7a3ba09c0297a31af215563ecf42817c98df621387f4e949"
dependencies = [
 "bitflags",
 "errno",
 "libc",
 "linux-raw-sys",
 "windows-sys",
]

[[package]]
name = "rusty-fork"
version = "0.3.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "cb3dcc6e454c328bb824492db107ab7c0ae8fcffe4ad210136ef014458c1bc4f"
dependencies = [
 "fnv",
 "quick-error",
 "tempfile",
 "wait-timeout",
]

[[package]]
name = "ryu"
version = "1.0.16"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f98d2aa92eebf49b69786be48e4477826b256916e84a57ff2a4f21923b48eb4c"

[[package]]
name = "sec1"
version = "0.7.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d3e97a565f76233a6003f9f5c54be1d9c5bdfa3eccfb189469f11ec4901c47dc"
dependencies = [
 "base16ct",
```

```
 "der",
 "generic-array",
 "pkcs8",
 "subtle",
 "zeroize",
]

[[package]]
name = "semver"
version = "0.11.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f301af10236f6df4160f7c3f04eec6dbc70ace82d23326abad5edee88801c6b6"
dependencies = [
 "semver-parser",
]

[[package]]
name = "semver"
version = "1.0.21"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b97ed7a9823b74f99c7742f5336af7be5ecd3eeafcb1507d1fa93347b1d589b0"

[[package]]
name = "semver-parser"
version = "0.10.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "00b0bef5b7f9e0df16536d3961cfb6e84331c065b4066afb39768d0e319411f7"
dependencies = [
 "pest",
]

[[package]]
name = "serde"
version = "1.0.196"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "870026e60fa08c69f064aa766c10f10b1d62db9ccd4d0abb206472bee0ce3b32"
dependencies = [
 "serde_derive",
]

[[package]]
name = "serde_derive"
version = "1.0.196"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "33c85360c95e7d137454dc81d9a4ed2b8efd8fbe19cee57357b32b9771fccb67"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "serde_json"
version = "1.0.113"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "69801b70b1c3dac963ecb03a364ba0ceda9cf60c71cfe475e99864759c8b8a79"
dependencies = [
 "itoa",
 "ryu",
 "serde",
]

[[package]]
name = "sha2"
version = "0.10.8"
source = "registry+https://github.com/rust-lang/crates.io-index"
```

```
checksum = "793db75ad2bcafc3ffa7c68b215fee268f537982cd901d132f89c6343f3a3dc8"
dependencies = [
 "cfg-if",
 "cpufeatures",
 "digest 0.10.7",
]

[[package]]
name = "sha3-asm"
version = "0.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bac61da6b35ad76b195eb4771210f947734321a8d81d7738e1580d953bc7a15e"
dependencies = [
 "cc",
 "cfg-if",
]

[[package]]
name = "signature"
version = "2.2.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "77549399552de45a898a580c1b41d445bf730df867cc44e6c0233bbc4b8329de"
dependencies = [
 "digest 0.10.7",
 "rand_core",
]

[[package]]
name = "slab"
version = "0.4.9"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8f92a496fb766b417c996b9c5e57daf2f7ad3b0bebe1ccfca4856390e3d3bb67"
dependencies = [
 "autocfg",
]

[[package]]
name = "smallvec"
version = "1.13.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "e6ecd384b10a64542d77071bd64bd7b231f4ed5940fba55e98c3de13824cf3d7"

[[package]]
name = "spdx"
version = "0.10.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "62bde1398b09b9f93fc2fc9b9da86e362693e999d3a54a8ac47a99a5a73f638b"
dependencies = [
 "smallvec",
]

[[package]]
name = "spki"
version = "0.7.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d91ed6c858b01f942cd56b37a94b3e0a1798290327d1236e4d9cf4eaca44d29d"
dependencies = [
 "base64ct",
 "der",
]

[[package]]
name = "static_assertions"
version = "1.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a2eb9349b6444b326872e140eb1cf5e7c522154d69e7a0ffb0fb81c06b37543f"
```

```
[[package]]
name = "subtle"
version = "2.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "81cdd64d312baedb58e21336b31bc043b77e01cc99033ce76ef539f78e965ebc"

[[package]]
name = "syn"
version = "1.0.109"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "72b64191b275b66ffe2469e8af2c1cfe3bafa67b529ead792a6d0160888b4237"
dependencies = [
 "proc-macro2",
 "quote",
 "unicode-ident",
]

[[package]]
name = "syn"
version = "2.0.48"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0f3531638e407dfc0814761abb7c00a5b54992b849452a0646b7f65c9f770f3f"
dependencies = [
 "proc-macro2",
 "quote",
 "unicode-ident",
]

[[package]]
name = "syn-solidity"
version = "0.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4497156948bd342b52038035a6fa514a89626e37af9d2c52a5e8d8ebcc7ee479"
dependencies = [
 "paste",
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "tap"
version = "1.0.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "55937e1799185b12863d447f42597ed69d9928686b8d88a1df17376a097d8369"

[[package]]
name = "tempfile"
version = "3.10.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a365e8cd18e44762ef95d87f284f4b5cd04107fec2ff3052bd6a3e6069669e67"
dependencies = [
 "cfg-if",
 "fastrand",
 "rustix",
 "windows-sys",
]

[[package]]
name = "thiserror"
version = "1.0.56"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d54378c645627613241d077a3a79db965db602882668f9136ac42af9ecb730ad"
dependencies = [
 "thiserror-impl",
```

```
]

[[package]]
name = "thiserror-impl"
version = "1.0.56"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "fa0faa943b50f3db30a20aa7e265dbc66076993efed8463e8de414e5d06d3471"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "threadpool"
version = "1.8.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d050e60b33d41c19108b32cea32164033a9013fe3b46cbd4457559bfbf77afaa"
dependencies = [
 "num_cpus",
]

[[package]]
name = "tiny-keccak"
version = "2.0.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "2c9d3793400a45f954c52e73d068316d76b6f4e36977e3fcebb13a2721e80237"
dependencies = [
 "crunchy",
]

[[package]]
name = "tinyvec"
version = "1.6.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "87cc5ceb3875bb20c2890005a4e226a4651264a5c75edb2421b52861a0a0cb50"
dependencies = [
 "tinyvec_macros",
]

[[package]]
name = "tinyvec_macros"
version = "0.1.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1f3ccbac311fea05f86f61904b462b55fb3df8837a366dfc601a0161d0532f20"

[[package]]
name = "tokio"
version = "1.36.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "61285f6515fa018fb2d1e46eb21223fff441ee8db5d0f1435e8ab4f5cdb80931"
dependencies = [
 "backtrace",
 "pin-project-lite",
]

[[package]]
name = "toml_datetime"
version = "0.6.3"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "7cda73e2f1397b1262d6dfdcef8aafae14d1de7748d66822d3bfeeb6d03e5e4b"

[[package]]
name = "toml_edit"
version = "0.20.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
```

```
checksum = "396e4d48bbb2b7554c944bde63101b5ae446cff6ec4a24227428f15eb72ef338"
dependencies = [
 "indexmap",
 "toml_datetime",
 "winnow",
]

[[package]]
name = "tower"
version = "0.4.13"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b8fa9be0de6cf49e536ce1851f987bd21a43b771b09473c3549a6c853db37c1c"
dependencies = [
 "futures-core",
 "futures-util",
 "pin-project",
 "pin-project-lite",
 "tower-layer",
 "tower-service",
 "tracing",
]

[[package]]
name = "tower-layer"
version = "0.3.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c20c8dbed6283a09604c3e69b4b7eeb54e298b8a600d4d5ecb5ad39de609f1d0"

[[package]]
name = "tower-service"
version = "0.3.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b6bc1c9ce2b5135ac7f93c72918fc37feb872bdc6a5533a8b85eb4b86bfdae52"

[[package]]
name = "tracing"
version = "0.1.40"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c3523ab5a71916ccf420eebdf5521fcef02141234bbc0b8a49f2fdc4544364ef"
dependencies = [
 "log",
 "pin-project-lite",
 "tracing-attributes",
 "tracing-core",
]

[[package]]
name = "tracing-attributes"
version = "0.1.27"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "34704c8d6ebcbc939824180af020566b01a7c01f80641264eba0999f6c2b6be7"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]

[[package]]
name = "tracing-core"
version = "0.1.32"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "c06d3da6113f116aaee68e4d601191614c9053067f9ab7f6edbcb161237daa54"
dependencies = [
 "once_cell",
]
```

```
[[package]]
name = "typenum"
version = "1.17.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "42ff0bf0c66b8238c6f3b578df37d0b7848e55df8577b3f74f92a69acceeb825"

[[package]]
name = "ucd-trie"
version = "0.1.6"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ed646292ffc8188ef8ea4d1e0e0150fb15a5c2e12ad9b8fc191ae7a8a7f3c4b9"

[[package]]
name = "uint"
version = "0.9.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "76f64bba2c53b04fcab63c01a7d7427eadc821e3bc48c34dc9ba29c501164b52"
dependencies = [
 "byteorder",
 "crunchy",
 "hex",
 "static_assertions",
]

[[package]]
name = "unarray"
version = "0.1.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "eaea85b334db583fe3274d12b4cd1880032beab409c0d774be044d4480ab9a94"

[[package]]
name = "unicase"
version = "2.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f7d2d4dafb69621809a81864c9c1b864479e1235c0dd4e199924b9742439ed89"
dependencies = [
 "version_check",
]

[[package]]
name = "unicode-bidi"
version = "0.3.15"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "08f95100a766bf4f8f28f90d77e0a5461bbdb219042e7679bebe79004fed8d75"

[[package]]
name = "unicode-ident"
version = "1.0.12"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3354b9ac3fae1ff6755cb6db53683adb661634f67557942dea4facebec0fee4b"

[[package]]
name = "unicode-normalization"
version = "0.1.22"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "5c5713f0fc4b5db668a2ac63cdb7bb4469d8c9fed047b1d0292cc7b0ce2ba921"
dependencies = [
 "tinyvec",
]

[[package]]
name = "unicode-segmentation"
version = "1.10.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1dd624098567895118886609431a7c3b8f516e41d30e0643f03d94592a147e36"
```

```
[[package]]
name = "unicode-xid"
version = "0.2.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f962df74c8c05a667b5ee8bcf162993134c104e96440b663c8daa176dc772d8c"

[[package]]
name = "url"
version = "2.5.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "31e6302e3bb753d46e83516cae55ae196fc0c309407cf11ab35cc51a4c2a4633"
dependencies = [
 "form_urlencoded",
 "idna",
 "percent-encoding",
]

[[package]]
name = "valuable"
version = "0.1.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "830b7e5d4d90034032940e4ace0d9a9a057e7a45cd94e6c007832e39edb82f6d"

[[package]]
name = "version_check"
version = "0.9.4"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "49874b5167b65d7193b8aba1567f5c7d93d001cafc34600cee003eda787e483f"

[[package]]
name = "wait-timeout"
version = "0.2.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9f200f5b12eb75f8c1ed65abd4b2db8a6e1b138a20de009dacee265a2498f3f6"
dependencies = [
 "libc",
]

[[package]]
name = "wasi"
version = "0.11.0+wasi-snapshot-preview1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9c8d87e72b64a3b4db28d11ce29237c246188f4f51057d65a7eab63b7987e423"

[[package]]
name = "wasm-bindgen"
version = "0.2.90"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "b1223296a201415c7fad14792dbefaace9bd52b62d33453ade1c5b5f07555406"
dependencies = [
 "cfg-if",
 "wasm-bindgen-macro",
]

[[package]]
name = "wasm-bindgen-backend"
version = "0.2.90"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "fcdc935b63408d58a32f8cc9738a0bffd8f05cc7c002086c6ef20b7312ad9dcd"
dependencies = [
 "bumpalo",
 "log",
 "once_cell",
 "proc-macro2",
 "quote",
 "syn 2.0.48",
```

```
 "wasm-bindgen-shared",
]

[[package]]
name = "wasm-bindgen-futures"
version = "0.4.40"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bde2032aeb86bdfaecc8b261eef3cba735cc426c1f3a3416d1e0791be95fc461"
dependencies = [
 "cfg-if",
 "js-sys",
 "wasm-bindgen",
 "web-sys",
]

[[package]]
name = "wasm-bindgen-macro"
version = "0.2.90"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3e4c238561b2d428924c49815533a8b9121c664599558a5d9ec51f8a1740a999"
dependencies = [
 "quote",
 "wasm-bindgen-macro-support",
]

[[package]]
name = "wasm-bindgen-macro-support"
version = "0.2.90"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bae1abb6806dc1ad9e560ed242107c0f6c84335f1749dd4e8ddb012ebd5e25a7"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
 "wasm-bindgen-backend",
 "wasm-bindgen-shared",
]

[[package]]
name = "wasm-bindgen-shared"
version = "0.2.90"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "4d91413b1c31d7539ba5ef2451af3f0b833a005eb27a631cec32bc0635a8602b"

[[package]]
name = "wasm-encoder"
version = "0.202.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bfd106365a7f5f7aa3c1916a98cbb3ad477f5ff96ddb130285a91c6e7429e67a"
dependencies = [
 "leb128",
]

[[package]]
name = "wasm-metadata"
version = "0.202.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "094aea3cb90e09f16ee25a4c0e324b3e8c934e7fd838bfa039aef5352f44a917"
dependencies = [
 "anyhow",
 "indexmap",
 "serde",
 "serde_derive",
 "serde_json",
 "spdx",
 "wasm-encoder",
```

```
 "wasmparser",
]

[[package]]
name = "wasmparser"
version = "0.202.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "d6998515d3cf3f8b980ef7c11b29a9b1017d4cf86b99ae93b546992df9931413"
dependencies = [
 "bitflags",
 "indexmap",
 "semver 1.0.21",
]

[[package]]
name = "web-sys"
version = "0.3.67"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "58cd2333b6e0be7a39605f0e255892fd7418a682d8da8fe042fe25128794d2ed"
dependencies = [
 "js-sys",
 "wasm-bindgen",
]

[[package]]
name = "windows-sys"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "282be5f36a8ce781fad8c8ae18fa3f9beff57ec1b52cb3de0789201425d9a33d"
dependencies = [
 "windows-targets",
]

[[package]]
name = "windows-targets"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "8a18201040b24831fbb9e4eb208f8892e1f50a37feb53cc7ff887feb8f50e7cd"
dependencies = [
 "windows_aarch64_gnullvm",
 "windows_aarch64_msvc",
 "windows_i686_gnu",
 "windows_i686_msvc",
 "windows_x86_64_gnu",
 "windows_x86_64_gnullvm",
 "windows_x86_64_msvc",
]

[[package]]
name = "windows_aarch64_gnullvm"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "cb7764e35d4db8a7921e09562a0304bf2f93e0a51bfccee0bd0bb0b666b015ea"

[[package]]
name = "windows_aarch64_msvc"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "bbaa0368d4f1d2aaefc55b6fcfee13f41544ddf36801e793edbbfd7d7df075ef"

[[package]]
name = "windows_i686_gnu"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a28637cb1fa3560a16915793afb20081aba2c92ee8af57b4d5f28e4b3e7df313"
```

```
[[package]]
name = "windows_i686_msvc"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ffe5e8e31046ce6230cc7215707b816e339ff4d4d67c65dffa206fd0f7aa7b9a"

[[package]]
name = "windows_x86_64_gnu"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3d6fa32db2bc4a2f5abeacf2b69f7992cd09dca97498da74a151a3132c26befd"

[[package]]
name = "windows_x86_64_gnullvm"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "1a657e1e9d3f514745a572a6846d3c7aa7dbe1658c056ed9c3344c4109a6949e"

[[package]]
name = "windows_x86_64_msvc"
version = "0.52.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "dff9641d1cd4be8d1a070daf9e3773c5f67e78b4d9d42263020c057706765c04"

[[package]]
name = "winnow"
version = "0.5.37"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a7cad8365489051ae9f054164e459304af2e7e9bb407c958076c8bf4aef52da5"
dependencies = [
 "memchr",
]

[[package]]
name = "wit-bindgen"
version = "0.24.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9fb4e7653763780be47e38f479e9aa83c768aa6a3b2ed086dc2826fdbbb7e7f5"
dependencies = [
 "wit-bindgen-rt",
 "wit-bindgen-rust-macro",
]

[[package]]
name = "wit-bindgen-core"
version = "0.24.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "9b67e11c950041849a10828c7600ea62a4077c01e8af72e8593253575428f91b"
dependencies = [
 "anyhow",
 "wit-parser",
]

[[package]]
name = "wit-bindgen-rt"
version = "0.24.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "3b0780cf7046630ed70f689a098cd8d56c5c3b22f2a7379bbdb088879963ff96"
dependencies = [
 "bitflags",
]

[[package]]
name = "wit-bindgen-rust"
version = "0.24.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
```

```
checksum = "30acbe8fb708c3a830a33c4cb705df82659bf831b492ec6ca1a17a369cfeeafb"
dependencies = [
 "anyhow",
 "heck 0.4.1",
 "indexmap",
 "wasm-metadata",
 "wit-bindgen-core",
 "wit-component",
]

[[package]]
name = "wit-bindgen-rust-macro"
version = "0.24.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "2b1b06eae85feaecdf9f2854f7cac124e00d5a6e5014bfb02eb1ecdeb5f265b9"
dependencies = [
 "anyhow",
 "proc-macro2",
 "quote",
 "syn 2.0.48",
 "wit-bindgen-core",
 "wit-bindgen-rust",
]

[[package]]
name = "wit-component"
version = "0.202.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "0c836b1fd9932de0431c1758d8be08212071b6bba0151f7bac826dbc4312a2a9"
dependencies = [
 "anyhow",
 "bitflags",
 "indexmap",
 "log",
 "serde",
 "serde_derive",
 "serde_json",
 "wasm-encoder",
 "wasm-metadata",
 "wasmparser",
 "wit-parser",
]

[[package]]
name = "wit-parser"
version = "0.202.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "744237b488352f4f27bca05a10acb79474415951c450e52ebd0da784c1df2bcc"
dependencies = [
 "anyhow",
 "id-arena",
 "indexmap",
 "log",
 "semver 1.0.21",
 "serde",
 "serde_derive",
 "serde_json",
 "unicode-xid",
 "wasmparser",
]

[[package]]
name = "wyz"
version = "0.5.1"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "05f360fc0b24296329c78fda852a1e9ae82de9cf7b27dae4b7f62f118f77b9ed"
```

```
dependencies = [
 "tap",
]

[[package]]
name = "zeroize"
version = "1.7.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "525b4ec142c6b68a2d10f01f7bbf6755599ca3f81ea53b8431b7dd348f5fdb2d"
dependencies = [
 "zeroize_derive",
]

[[package]]
name = "zeroize_derive"
version = "1.4.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "ce36e65b0d2999d2aafac989fb249189a141aee1f53c612c1f37d72631959f69"
dependencies = [
 "proc-macro2",
 "quote",
 "syn 2.0.48",
]
```

Cargo.toml
==========

```
[package]
name = "kinode_process_lib"
description = "A library for writing Kinode processes in Rust."
version = "0.7.0"
edition = "2021"
license-file = "LICENSE"
homepage = "https://kinode.org"
repository = "https://github.com/kinode-dao/process_lib"

[dependencies]
alloy-rpc-types = { git = "https://github.com/alloy-rs/alloy", rev = "cad7935" }
alloy-primitives = "0.7.0"
alloy-transport = { git = "https://github.com/alloy-rs/alloy.git", rev = "cad7935" }
alloy-json-rpc = { git = "https://github.com/alloy-rs/alloy.git", rev = "cad7935" }
anyhow = "1.0"
bincode = "1.3.3"
http = "1.0.0"
mime_guess = "2.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
rand = "0.8"
rmp-serde = "1.1.2"
thiserror = "1.0"
url = "2.4.1"
wit-bindgen = "0.24.0"
```

pull_request_template.md
========================

```
## Problem

{A brief description of the problem, along with necessary context.}

## Solution

{A brief description of how you solved the problem.}

## Docs Update
```

[Corresponding docs PR](https://github.com/kinode-dao/kinode-book/pull/my-pr-number)

Notes

{Any other information useful for reviewers.}

README.md
=========

process_lib

Library of functions for more ergonomic kinode process development.

To develop/build:
```
git submodule update --init
```

Docs: (TODO link)

kinode-wit/.git
===============

gitdir: ../.git/modules/kinode-wit

kinode-wit/kinode.wit
=====================

```
package kinode:process@0.7.0;

interface standard {
    //
    // System types:
    //

    // JSON is passed over WASM boundary as a string.
    type json = string;

    type node-id = string;

    // Context, like a message body, is a protocol-defined serialized byte
    // array. It is used when building a Request to save information that
    // will not be part of a Response, in order to more easily handle
    // ("contextualize") that Response.
    type context = list<u8>;

    record process-id {
        process-name: string,
        package-name: string,
        publisher-node: node-id,
    }

    record address {
        node: node-id,
        process: process-id,
    }

    record lazy-load-blob {
        mime: option<string>,
        bytes: list<u8>,
    }

    record request {
        // set in order to inherit lazy-load-blob from parent message, and if
        // expects-response is none, direct response to source of parent.
        // also carries forward certain aspects of parent message in kernel,
```

```
    // see documentation for formal spec and examples.
    inherit: bool,
    // if some, request expects a response in the given number of seconds
    expects-response: option<u64>,
    body: list<u8>,
    metadata: option<json>,
    capabilities: list<capability>,
    // to grab lazy-load-blob, use get_blob()
}

record response {
    inherit: bool,
    body: list<u8>,
    metadata: option<json>,
    capabilities: list<capability>,
    // to grab lazy-load-blob, use get_blob()
}

// A message can be a request or a response. within a response, there is
// a result which surfaces any error that happened because of a request.
// A successful response will contain the context of the request it
// matches, if any was set.
variant message {
    request(request),
    response(tuple<response, option<context>>),
}

record capability {
    issuer: address,
    params: json,
}

// On-exit is a setting that determines what happens when a process
// panics, completes, or otherwise "ends". NOTE: requests should have
// expects-response set to false, will always be set to that by kernel.
variant on-exit {
    none,
    restart,
    requests(list<tuple<address, request, option<lazy-load-blob>>>),
}

// Network errors come from trying to send a message to another node.
// A message can fail by timing out, or by the node being entirely
// unreachable (offline). In either case, the message is not delivered
// and the process that sent it receives that message along with any
// assigned context and/or lazy-load-blob, and is free to handle it as it
// sees fit. Note that if the message is a response, the process can
// issue a response again, and it will be directed to the same (remote)
// request as the original.
record send-error {
    kind: send-error-kind,
    message: message,
    lazy-load-blob: option<lazy-load-blob>,
}

enum send-error-kind {
    offline,
    timeout,
}

enum spawn-error {
    name-taken,
    no-file-at-path,
    // TODO more here?
}
```

```
//
// System utils:
//

print-to-terminal: func(verbosity: u8, message: string);

//
// Process management:
//

set-on-exit: func(on-exit: on-exit);

get-on-exit: func() -> on-exit;

get-state: func() -> option<list<u8>>;

set-state: func(bytes: list<u8>);

clear-state: func();

spawn: func(
    name: option<string>,
    wasm-path: string, // must be located within package's drive
    on-exit: on-exit,
    request-capabilities: list<capability>,
    // note that we are restricting granting to just messaging the
    // newly spawned process
    grant-capabilities: list<process-id>,
    public: bool
) -> result<process-id, spawn-error>;

//
// Capabilities management:
//

// Saves the capabilities to persisted process state.
save-capabilities: func(caps: list<capability>);

// Deletes the capabilities from persisted process state.
drop-capabilities: func(caps: list<capability>);

// Gets all capabilities from persisted process state.
our-capabilities: func() -> list<capability>;

//
// Message I/O:
//

// Ingest next message when it arrives along with its source.
// Almost all long-running processes will call this in a loop.
receive: func() ->
    result<tuple<address, message>, tuple<send-error, option<context>>>;

// Gets lazy-load-blob, if any, of the message we most recently received.
get-blob: func() -> option<lazy-load-blob>;

// Send message(s) to target(s).
send-request: func(
    target: address,
    request: request,
    context: option<context>,
    lazy-load-blob: option<lazy-load-blob>
);

send-requests: func(
    requests: list<tuple<address,
```

```
                request,
                option<context>,
                option<lazy-load-blob>>>
    );

    send-response: func(
        response: response,
        lazy-load-blob: option<lazy-load-blob>
    );

    // Send a single request, then block (internally) until its response. The
    // type returned is Message but will always contain Response.
    send-and-await-response: func(
        target: address,
        request: request,
        lazy-load-blob: option<lazy-load-blob>
    ) -> result<tuple<address, message>, send-error>;
}

world lib {
    import standard;
}

world process {
    include lib;

    export init: func(our: string);
}
```

src/sqlite.rs
==========

```
use crate::{get_blob, Message, PackageId, Request};
use serde::{Deserialize, Serialize};
use std::collections::HashMap;
use thiserror::Error;

/// Actions are sent to a specific sqlite database, "db" is the name,
/// "package_id" is the package. Capabilities are checked, you can access another process's
/// database if it has given you the capability.
#[derive(Debug, Serialize, Deserialize)]
pub struct SqliteRequest {
    pub package_id: PackageId,
    pub db: String,
    pub action: SqliteAction,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum SqliteAction {
    Open,
    RemoveDb,
    Write {
        statement: String,
        tx_id: Option<u64>,
    },
    Read {
        query: String,
    },
    BeginTx,
    Commit {
        tx_id: u64,
    },
    Backup,
}

#[derive(Debug, Serialize, Deserialize)]
```

```
pub enum SqliteResponse {
    Ok,
    Read,
    BeginTx { tx_id: u64 },
    Err { error: SqliteError },
}

#[derive(Debug, Clone, Serialize, Deserialize, PartialEq)]
pub enum SqlValue {
    Integer(i64),
    Real(f64),
    Text(String),
    Blob(Vec<u8>),
    Boolean(bool),
    Null,
}

#[derive(Debug, Serialize, Deserialize, Error)]
pub enum SqliteError {
    #[error("sqlite: DbDoesNotExist")]
    NoDb,
    #[error("sqlite: NoTx")]
    NoTx,
    #[error("sqlite: No capability: {error}")]
    NoCap { error: String },
    #[error("sqlite: UnexpectedResponse")]
    UnexpectedResponse,
    #[error("sqlite: NotAWriteKeyword")]
    NotAWriteKeyword,
    #[error("sqlite: NotAReadKeyword")]
    NotAReadKeyword,
    #[error("sqlite: Invalid Parameters")]
    InvalidParameters,
    #[error("sqlite: IO error: {error}")]
    IOError { error: String },
    #[error("sqlite: rusqlite error: {error}")]
    RusqliteError { error: String },
    #[error("sqlite: input bytes/json/key error: {error}")]
    InputError { error: String },
}

/// Sqlite helper struct for a db.
/// Opening or creating a db will give you a Result<sqlite>.
/// You can call it's impl functions to interact with it.
pub struct Sqlite {
    pub package_id: PackageId,
    pub db: String,
    pub timeout: u64,
}

impl Sqlite {
    /// Query database. Only allows sqlite read keywords.
    pub fn read(
        &self,
        query: String,
        params: Vec<serde_json::Value>,
    ) -> anyhow::Result<Vec<HashMap<String, serde_json::Value>>> {
        let res = Request::new()
            .target(("our", "sqlite", "distro", "sys"))
            .body(serde_json::to_vec(&SqliteRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: SqliteAction::Read { query },
            })?)
            .blob_bytes(serde_json::to_vec(&params)?)
            .send_and_await_response(self.timeout)?;
```

```
    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<SqliteResponse>(&body)?;

            match response {
                SqliteResponse::Read => {
                    let blob = get_blob().ok_or_else(|| SqliteError::InputError {
                        error: "sqlite: no blob".to_string(),
                    })?;
                    let values = serde_json::from_slice::<
                        Vec<HashMap<String, serde_json::Value>>,
                    >(&blob.bytes)
                    .map_err(|e| SqliteError::InputError {
                        error: format!("sqlite: gave unparsable response: {}", e),
                    })?;
                    Ok(values)
                }
                SqliteResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!(
                    "sqlite: unexpected response {:?}",
                    response
                )),
            }
        }
        _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
    }
}

/// Execute a statement. Only allows sqlite write keywords.
pub fn write(
    &self,
    statement: String,
    params: Vec<serde_json::Value>,
    tx_id: Option<u64>,
) -> anyhow::Result<()> {
    let res = Request::new()
        .target(("our", "sqlite", "distro", "sys"))
        .body(serde_json::to_vec(&SqliteRequest {
            package_id: self.package_id.clone(),
            db: self.db.clone(),
            action: SqliteAction::Write { statement, tx_id },
        })?)
        .blob_bytes(serde_json::to_vec(&params)?)
        .send_and_await_response(self.timeout)?;

    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<SqliteResponse>(&body)?;

            match response {
                SqliteResponse::Ok => Ok(()),
                SqliteResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!(
                    "sqlite: unexpected response {:?}",
                    response
                )),
            }
        }
        _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
    }
}

/// Begin a transaction.
pub fn begin_tx(&self) -> anyhow::Result<u64> {
    let res = Request::new()
```

```
            .target(("our", "sqlite", "distro", "sys"))
            .body(serde_json::to_vec(&SqliteRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: SqliteAction::BeginTx,
            })?)
            .send_and_await_response(self.timeout)?;

        match res {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<SqliteResponse>(&body)?;

                match response {
                    SqliteResponse::BeginTx { tx_id } => Ok(tx_id),
                    SqliteResponse::Err { error } => Err(error.into()),
                    _ => Err(anyhow::anyhow!(
                        "sqlite: unexpected response {:?}",
                        response
                    )),
                }
            }
            _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
        }
    }

    /// Commit a transaction.
    pub fn commit_tx(&self, tx_id: u64) -> anyhow::Result<()> {
        let res = Request::new()
            .target(("our", "sqlite", "distro", "sys"))
            .body(serde_json::to_vec(&SqliteRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: SqliteAction::Commit { tx_id },
            })?)
            .send_and_await_response(self.timeout)?;

        match res {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<SqliteResponse>(&body)?;

                match response {
                    SqliteResponse::Ok => Ok(()),
                    SqliteResponse::Err { error } => Err(error.into()),
                    _ => Err(anyhow::anyhow!(
                        "sqlite: unexpected response {:?}",
                        response
                    )),
                }
            }
            _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
        }
    }
}

/// Open or create sqlite database.
pub fn open(package_id: PackageId, db: &str, timeout: Option<u64>) -> anyhow::Result<Sqlite> {
    let timeout = timeout.unwrap_or(5);

    let res = Request::new()
        .target(("our", "sqlite", "distro", "sys"))
        .body(serde_json::to_vec(&SqliteRequest {
            package_id: package_id.clone(),
            db: db.to_string(),
            action: SqliteAction::Open,
        })?)
        .send_and_await_response(timeout)?;
```

```
    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<SqliteResponse>(&body)?;

            match response {
                SqliteResponse::Ok => Ok(Sqlite {
                    package_id,
                    db: db.to_string(),
                    timeout,
                }),
                SqliteResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!(
                    "sqlite: unexpected response {:?}",
                    response
                )),
            }
        }
        _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
    }
}

/// Remove and delete sqlite database.
pub fn remove_db(package_id: PackageId, db: &str, timeout: Option<u64>) -> anyhow::Result<()> {
    let timeout = timeout.unwrap_or(5);

    let res = Request::new()
        .target(("our", "sqlite", "distro", "sys"))
        .body(serde_json::to_vec(&SqliteRequest {
            package_id: package_id.clone(),
            db: db.to_string(),
            action: SqliteAction::RemoveDb,
        })?)
        .send_and_await_response(timeout)?;

    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<SqliteResponse>(&body)?;

            match response {
                SqliteResponse::Ok => Ok(()),
                SqliteResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!(
                    "sqlite: unexpected response {:?}",
                    response
                )),
            }
        }
        _ => Err(anyhow::anyhow!("sqlite: unexpected message: {:?}", res)),
    }
}

src/kernel_types.rs
===================

use crate::kinode::process::standard as wit;
use crate::{Address, ProcessId};
use serde::{Deserialize, Serialize};
use std::collections::{HashMap, HashSet};

//
// process-facing kernel types, used for process
// management and message-passing
// matches types in kinode.wit
//
```

```
pub type Context = Vec<u8>;
pub type NodeId = String; // QNS domain name

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct LazyLoadBlob {
    pub mime: Option<String>, // MIME type
    pub bytes: Vec<u8>,
}

#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct Request {
    pub inherit: bool,
    pub expects_response: Option<u64>, // number of seconds until timeout
    pub body: Vec<u8>,
    pub metadata: Option<String>, // JSON-string
    pub capabilities: Vec<Capability>,
}

#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct Response {
    pub inherit: bool,
    pub body: Vec<u8>,
    pub metadata: Option<String>, // JSON-string
    pub capabilities: Vec<Capability>,
}

#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub enum Message {
    Request(Request),
    Response((Response, Option<Context>)),
}

#[derive(Clone, Debug, Eq, Hash, PartialEq, Serialize, Deserialize)]
pub struct Capability {
    pub issuer: Address,
    pub params: String, // JSON-string
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct SendError {
    pub kind: SendErrorKind,
    pub target: Address,
    pub message: Message,
    pub lazy_load_blob: Option<LazyLoadBlob>,
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum SendErrorKind {
    Offline,
    Timeout,
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum OnExit {
    None,
    Restart,
    Requests(Vec<(Address, Request, Option<LazyLoadBlob>)>),
}

impl OnExit {
    pub fn is_restart(&self) -> bool {
        match self {
            OnExit::None => false,
            OnExit::Restart => true,
            OnExit::Requests(_) => false,
        }
```

```
    }
}

/// IPC body format for requests sent to kernel runtime module
#[derive(Debug, Serialize, Deserialize)]
pub enum KernelCommand {
    /// RUNTIME ONLY: used to notify the kernel that booting is complete and
    /// all processes have been loaded in from their persisted or bootstrapped state.
    Booted,
    /// Tell the kernel to install and prepare a new process for execution.
    /// The process will not begin execution until the kernel receives a
    /// `RunProcess` command with the same `id`.
    ///
    /// The process that sends this command will be given messaging capabilities
    /// for the new process if `public` is false.
    ///
    /// All capabilities passed into initial_capabilities must be held by the source
    /// of this message, or the kernel will discard them (silently for now).
    InitializeProcess {
        id: ProcessId,
        wasm_bytes_handle: String,
        wit_version: Option<u32>,
        on_exit: OnExit,
        initial_capabilities: HashSet<Capability>,
        public: bool,
    },
    /// Create an arbitrary capability and grant it to a process.
    GrantCapabilities {
        target: ProcessId,
        capabilities: Vec<Capability>,
    },
    /// Drop capabilities. Does nothing if process doesn't have these caps
    DropCapabilities {
        target: ProcessId,
        capabilities: Vec<Capability>,
    },
    /// Tell the kernel to run a process that has already been installed.
    /// TODO: in the future, this command could be extended to allow for
    /// resource provision.
    RunProcess(ProcessId),
    /// Kill a running process immediately. This may result in the dropping / mishandling of messages!
    KillProcess(ProcessId),
    /// RUNTIME ONLY: notify the kernel that the runtime is shutting down and it
    /// should gracefully stop and persist the running processes.
    Shutdown,
    /// Ask kernel to produce debugging information
    Debug(KernelPrint),
}

#[derive(Debug, Serialize, Deserialize)]
pub enum KernelPrint {
    ProcessMap,
    Process(ProcessId),
    HasCap { on: ProcessId, cap: Capability },
}

/// IPC body format for all KernelCommand responses
#[derive(Debug, Serialize, Deserialize)]
pub enum KernelResponse {
    InitializedProcess,
    InitializeProcessError,
    StartedProcess,
    RunProcessError,
    KilledProcess(ProcessId),
}
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct PersistedProcess {
    pub wasm_bytes_handle: String,
    // pub drive: String,
    // pub full_path: String,
    pub on_exit: OnExit,
    pub capabilities: HashSet<Capability>,
    pub public: bool, // marks if a process allows messages from any process
}

#[derive(Serialize, Deserialize, Debug)]
pub enum StateAction {
    GetState(ProcessId),
    SetState(ProcessId),
    DeleteState(ProcessId),
    Backup,
}

#[derive(Serialize, Deserialize, Debug)]
pub enum StateResponse {
    GetState,
    SetState,
    DeleteState,
    Backup,
    Err(StateError),
}

#[derive(Debug, Serialize, Deserialize)]
pub enum StateError {
    RocksDBError { action: String, error: String },
    StartupError { action: String },
    BadBytes { action: String },
    BadRequest { error: String },
    BadJson { error: String },
    NotFound { process_id: ProcessId },
    IOError { error: String },
}

#[allow(dead_code)]
impl StateError {
    pub fn kind(&self) -> &str {
        match *self {
            StateError::RocksDBError { .. } => "RocksDBError",
            StateError::StartupError { .. } => "StartupError",
            StateError::BadBytes { .. } => "BadBytes",
            StateError::BadRequest { .. } => "BadRequest",
            StateError::BadJson { .. } => "NoJson",
            StateError::NotFound { .. } => "NotFound",
            StateError::IOError { .. } => "IOError",
        }
    }
}

//
// package types
//

/// Represents the metadata associated with a kinode package, which is an ERC721 compatible token.
/// This is deserialized from the `metadata.json` file in a package.
/// Fields:
/// - `name`: An optional field representing the display name of the package. This does not have to be uni
/// - `description`: An optional field providing a description of the package.
/// - `image`: An optional field containing a URL to an image representing the package.
/// - `external_url`: An optional field containing a URL for more information about the package. For examp
/// - `animation_url`: An optional field containing a URL to an animation or video representing the package
/// - `properties`: A requried field containing important information about the package.
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct Erc721Metadata {
    pub name: Option<String>,
    pub description: Option<String>,
    pub image: Option<String>,
    pub external_url: Option<String>,
    pub animation_url: Option<String>,
    pub properties: Erc721Properties,
}

/// Represents critical fields of a kinode package in an ERC721 compatible format.
/// This follows the [ERC1155](https://github.com/ethereum/ercs/blob/master/ERCS/erc-1155.md#erc-115
///
/// Fields:
/// - `package_name`: The unique name of the package, used in the `PackageId`, e.g. `package_name:pu
/// - `publisher`: The KNS identity of the package publisher used in the `PackageId`, e.g. `package_name
/// - `current_version`: A string representing the current version of the package, e.g. `1.0.0`.
/// - `mirrors`: A list of NodeIds where the package can be found, providing redundancy.
/// - `code_hashes`: A map from version names to their respective SHA-256 hashes.
/// - `license`: An optional field containing the license of the package.
/// - `screenshots`: An optional field containing a list of URLs to screenshots of the package.
/// - `wit_version`: An optional field containing the version of the WIT standard that the package adheres t
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct Erc721Properties {
    pub package_name: String,
    pub publisher: String,
    pub current_version: String,
    pub mirrors: Vec<NodeId>,
    pub code_hashes: HashMap<String, String>,
    pub license: Option<String>,
    pub screenshots: Option<Vec<String>>,
    pub wit_version: Option<(u32, u32, u32)>,
    pub dependencies: Vec<String>,
}

/// the type that gets deserialized from each entry in the array in `manifest.json`
#[derive(Debug, Serialize, Deserialize)]
pub struct PackageManifestEntry {
    pub process_name: String,
    pub process_wasm_path: String,
    pub on_exit: OnExit,
    pub request_networking: bool,
    pub request_capabilities: Vec<serde_json::Value>,
    pub grant_capabilities: Vec<serde_json::Value>,
    pub public: bool,
}

/// the type that gets deserialized from a `scripts.json` object
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct DotScriptsEntry {
    pub root: bool,
    pub public: bool,
    pub request_networking: bool,
    pub request_capabilities: Option<Vec<serde_json::Value>>,
    pub grant_capabilities: Option<Vec<serde_json::Value>>,
}

impl std::fmt::Display for Message {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match self {
            Message::Request(request) => write!(
                f,
                "Request(\n        inherit: {},\n        expects_response: {:?},\n        body: {} bytes,\n        metadata
                request.inherit,
                request.expects_response,
                request.body.len(),
```

```
                &request.metadata.as_ref().unwrap_or(&"None".into()),
            ),
            Message::Response((response, context)) => write!(
                f,
                "Response(\n    inherit: {},\n    body: {} bytes,\n    metadata: {},\n    context: {} bytes\n
                response.inherit,
                response.body.len(),
                &response.metadata.as_ref().unwrap_or(&"None".into()),
                if context.is_none() {
                    0
                } else {
                    context.as_ref().unwrap().len()
                },
            ),
        }
    }
}

//
// conversions between wit types and kernel types (annoying!)
//

pub fn de_wit_address(wit: wit::Address) -> Address {
    Address {
        node: wit.node,
        process: wit.process,
    }
}

pub fn en_wit_address(address: Address) -> wit::Address {
    wit::Address {
        node: address.node,
        process: address.process,
    }
}

pub fn de_wit_request(wit: wit::Request) -> Request {
    Request {
        inherit: wit.inherit,
        expects_response: wit.expects_response,
        body: wit.body,
        metadata: wit.metadata,
        capabilities: wit
            .capabilities
            .into_iter()
            .map(de_wit_capability)
            .collect(),
    }
}

pub fn en_wit_request(request: Request) -> wit::Request {
    wit::Request {
        inherit: request.inherit,
        expects_response: request.expects_response,
        body: request.body,
        metadata: request.metadata,
        capabilities: request
            .capabilities
            .into_iter()
            .map(en_wit_capability)
            .collect(),
    }
}

pub fn de_wit_response(wit: wit::Response) -> Response {
    Response {
```

```
        inherit: wit.inherit,
        body: wit.body,
        metadata: wit.metadata,
        capabilities: wit
            .capabilities
            .into_iter()
            .map(de_wit_capability)
            .collect(),
    }
}

pub fn en_wit_response(response: Response) -> wit::Response {
    wit::Response {
        inherit: response.inherit,
        body: response.body,
        metadata: response.metadata,
        capabilities: response
            .capabilities
            .into_iter()
            .map(en_wit_capability)
            .collect(),
    }
}

pub fn de_wit_blob(wit: Option<wit::LazyLoadBlob>) -> Option<LazyLoadBlob> {
    match wit {
        None => None,
        Some(wit) => Some(LazyLoadBlob {
            mime: wit.mime,
            bytes: wit.bytes,
        }),
    }
}

pub fn en_wit_blob(load: Option<LazyLoadBlob>) -> Option<wit::LazyLoadBlob> {
    match load {
        None => None,
        Some(load) => Some(wit::LazyLoadBlob {
            mime: load.mime,
            bytes: load.bytes,
        }),
    }
}

pub fn de_wit_capability(wit: wit::Capability) -> Capability {
    Capability {
        issuer: Address {
            node: wit.issuer.node,
            process: ProcessId {
                process_name: wit.issuer.process.process_name,
                package_name: wit.issuer.process.package_name,
                publisher_node: wit.issuer.process.publisher_node,
            },
        },
        params: wit.params,
    }
}

pub fn en_wit_capability(cap: Capability) -> wit::Capability {
    wit::Capability {
        issuer: en_wit_address(cap.issuer),
        params: cap.params,
    }
}

pub fn en_wit_message(message: Message) -> wit::Message {
```

```
    match message {
        Message::Request(request) => wit::Message::Request(en_wit_request(request)),
        Message::Response((response, context)) => {
            wit::Message::Response((en_wit_response(response), context))
        }
    }
}

pub fn en_wit_send_error(error: SendError) -> wit::SendError {
    wit::SendError {
        kind: en_wit_send_error_kind(error.kind),
        message: en_wit_message(error.message),
        lazy_load_blob: en_wit_blob(error.lazy_load_blob),
    }
}

pub fn en_wit_send_error_kind(kind: SendErrorKind) -> wit::SendErrorKind {
    match kind {
        SendErrorKind::Offline => wit::SendErrorKind::Offline,
        SendErrorKind::Timeout => wit::SendErrorKind::Timeout,
    }
}

#[derive(Debug, Serialize, Deserialize)]
pub enum MessageType {
    Request,
    Response,
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct KnsUpdate {
    pub name: String, // actual username / domain name
    pub owner: String,
    pub node: String, // hex namehash of node
    pub public_key: String,
    pub ip: String,
    pub port: u16,
    pub routers: Vec<String>,
}
```

src/capability.rs
==========

```
pub use crate::{Address, Capability};
use serde::de::{self, Deserialize, Deserializer, MapAccess, SeqAccess, Visitor};
use serde::ser::{Serialize, SerializeStruct};
use std::hash::{Hash, Hasher};

/// Capability is defined in the wit bindings, but constructors and methods here.
/// A `Capability` is a combination of an Address and a set of Params (a serialized
/// json string). Capabilities are attached to messages to either share that capability
/// with the receiving process, or to prove that a process has authority to perform a
/// certain action.
impl Capability {
    /// Create a new `Capability`. Takes a node ID and a process ID.
    pub fn new<T, U>(address: T, params: U) -> Capability
    where
        T: Into<Address>,
        U: Into<String>,
    {
        Capability {
            issuer: address.into(),
            params: params.into(),
        }
    }
    /// Read the node ID from a `Capability`.
```

```
    pub fn issuer(&self) -> &Address {
        &self.issuer
    }
    /// Read the params from a `Capability`.
    pub fn params(&self) -> &str {
        &self.params
    }
}

impl Serialize for Capability {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::ser::Serializer,
    {
        let mut state = serializer.serialize_struct("Capability", 2)?;
        state.serialize_field("issuer", &self.issuer)?;
        state.serialize_field("params", &self.params)?;
        state.end()
    }
}

impl<'a> Deserialize<'a> for Capability {
    fn deserialize<D>(deserializer: D) -> Result<Capability, D::Error>
    where
        D: serde::de::Deserializer<'a>,
    {
        enum Field {
            Issuer,
            Params,
        }

        impl<'de> Deserialize<'de> for Field {
            fn deserialize<D>(deserializer: D) -> Result<Field, D::Error>
            where
                D: Deserializer<'de>,
            {
                struct FieldVisitor;

                impl<'de> Visitor<'de> for FieldVisitor {
                    type Value = Field;

                    fn expecting(&self, formatter: &mut std::fmt::Formatter) -> std::fmt::Result {
                        formatter.write_str("`issuer` or `params`")
                    }

                    fn visit_str<E>(self, value: &str) -> Result<Field, E>
                    where
                        E: de::Error,
                    {
                        match value {
                            "issuer" => Ok(Field::Issuer),
                            "params" => Ok(Field::Params),
                            _ => Err(de::Error::unknown_field(value, FIELDS)),
                        }
                    }
                }

                deserializer.deserialize_identifier(FieldVisitor)
            }
        }

        struct CapabilityVisitor;

        impl<'de> Visitor<'de> for CapabilityVisitor {
            type Value = Capability;
```

```
            fn expecting(&self, formatter: &mut std::fmt::Formatter) -> std::fmt::Result {
                formatter.write_str("struct Capability")
            }

            fn visit_seq<V>(self, mut seq: V) -> Result<Capability, V::Error>
            where
                V: SeqAccess<'de>,
            {
                let issuer: Address = seq
                    .next_element()?
                    .ok_or_else(|| de::Error::invalid_length(0, &self))?;
                let params: String = seq
                    .next_element()?
                    .ok_or_else(|| de::Error::invalid_length(1, &self))?;
                Ok(Capability::new(issuer, params))
            }

            fn visit_map<V>(self, mut map: V) -> Result<Capability, V::Error>
            where
                V: MapAccess<'de>,
            {
                let mut issuer: Option<Address> = None;
                let mut params: Option<String> = None;
                while let Some(key) = map.next_key()? {
                    match key {
                        Field::Issuer => {
                            if issuer.is_some() {
                                return Err(de::Error::duplicate_field("issuer"));
                            }
                            issuer = Some(map.next_value()?);
                        }
                        Field::Params => {
                            if params.is_some() {
                                return Err(de::Error::duplicate_field("params"));
                            }
                            params = Some(map.next_value()?);
                        }
                    }
                }
                let issuer: Address = issuer
                    .ok_or_else(|| de::Error::missing_field("issuer"))?
                    .into();
                let params: String = params
                    .ok_or_else(|| de::Error::missing_field("params"))?
                    .into();
                Ok(Capability::new(issuer, params))
            }
        }

        const FIELDS: &'static [&'static str] = &["issuer", "params"];
        deserializer.deserialize_struct("Capability", FIELDS, CapabilityVisitor)
    }
}

impl Hash for Capability {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.issuer.hash(state);
        self.params.hash(state);
    }
}

impl Eq for Capability {}

impl PartialEq for Capability {
    fn eq(&self, other: &Self) -> bool {
        self.issuer == other.issuer && self.params == other.params
```

```
    }
}

impl From<&Capability> for Capability {
    fn from(input: &Capability) -> Self {
        input.clone()
    }
}

impl<T> From<(T, &str)> for Capability
where
    T: Into<Address>,
{
    fn from(input: (T, &str)) -> Self {
        Capability::new(input.0, input.1)
    }
}

impl std::fmt::Display for Capability {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}({})", self.issuer, self.params)
    }
}
```

src/timer.rs
==========

```
use crate::*;
use anyhow::Result;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum TimerAction {
    Debug,
    SetTimer(u64),
}

/// Set a timer using the runtime that will return a Response after the specified duration.
/// The duration should be a number of milliseconds.
pub fn set_timer(duration: u64, context: Option<Context>) {
    match context {
        None => {
            Request::new()
                .target(Address::new(
                    "our",
                    ProcessId::new(Some("timer"), "distro", "sys"),
                ))
                .body(serde_json::to_vec(&TimerAction::SetTimer(duration)).unwrap())
                .expects_response((duration / 1000) + 1)
                // safe to unwrap this call when we know we've set both target and body
                .send()
                .unwrap();
        }
        Some(context) => {
            Request::new()
                .target(Address::new(
                    "our",
                    ProcessId::new(Some("timer"), "distro", "sys"),
                ))
                .body(serde_json::to_vec(&TimerAction::SetTimer(duration)).unwrap())
                .expects_response((duration / 1000) + 1)
                .context(context)
                // safe to unwrap this call when we know we've set both target and body
                .send()
                .unwrap();
        }
    }
```

```
}

/// Set a timer using the runtime that will return a Response after the specified duration,
/// then wait for that timer to resolve. The duration should be a number of milliseconds.
pub fn set_and_await_timer(duration: u64) -> Result<Message, SendError> {
    Request::new()
        .target(Address::new(
            "our",
            ProcessId::new(Some("timer"), "distro", "sys"),
        ))
        .body(serde_json::to_vec(&TimerAction::SetTimer(duration)).unwrap())
        // safe to unwrap this call when we know we've set both target and body
        .send_and_await_response((duration / 1000) + 1)
        .unwrap()
}
```

src/response.rs
==========

```
use crate::*;

/// Response builder. Use [`Response::new()`] to start a response, then build it,
/// then call [`Response::send()`] on it to fire.
pub struct Response {
    inherit: bool,
    body: Option<Vec<u8>>,
    metadata: Option<String>,
    blob: Option<LazyLoadBlob>,
    capabilities: Vec<Capability>,
}

#[allow(dead_code)]
impl Response {
    /// Start building a new response. Attempting to send this response will
    /// not succeed until its `body` has been set with `body()` or `try_body()`.
    pub fn new() -> Self {
        Response {
            inherit: false,
            body: None,
            metadata: None,
            blob: None,
            capabilities: vec![],
        }
    }
    /// Set whether this response will "inherit" the blob of the request
    /// that this process most recently received. Unlike with requests, the
    /// inherit field of a response only deals with blob attachment, since
    /// responses don't themselves have to consider responses or contexts.
    ///
    /// *Note that if the blob is set for this response, this flag will not
    /// override it.*
    pub fn inherit(mut self, inherit: bool) -> Self {
        self.inherit = inherit;
        self
    }
    /// Set the IPC body (Inter-Process Communication) value for this message. This field
    /// is mandatory. An IPC body is simply a vector of bytes. Process developers are
    /// responsible for architecting the serialization/derserialization strategy
    /// for these bytes, but the simplest and most common strategy is just to use
    /// a JSON spec that gets stored in bytes as a UTF-8 string.
    ///
    /// If the serialization strategy is complex, it's best to define it as an impl
    /// of [`TryInto`] on your IPC body type, then use `try_body()` instead of this.
    pub fn body<T>(mut self, body: T) -> Self
    where
        T: Into<Vec<u8>>,
```

```
{
    self.body = Some(body.into());
    self
}
/// Set the IPC body (Inter-Process Communication) value for this message, using a
/// type that's got an implementation of [`TryInto`] for `Vec<u8>`. It's best
/// to define an IPC body type within your app, then implement TryFrom/TryInto for
/// all IPC body serialization/deserialization.
pub fn try_body<T>(mut self, body: T) -> anyhow::Result<Self>
where
    T: TryInto<Vec<u8>, Error = anyhow::Error>,
{
    self.body = Some(body.try_into()?);
    Ok(self)
}
/// Set the metdata field for this response. Metadata is simply a [`String`].
/// Metadata should usually be used for middleware and other message-passing
/// situations that require the original IPC body and blob to be preserved.
/// As such, metadata should not always be expected to reach the final destination
/// of this response unless the full chain of behavior is known / controlled by
/// the developer.
pub fn metadata(mut self, metadata: &str) -> Self {
    self.metadata = Some(metadata.to_string());
    self
}
/// Set the blob of this response. A [`LazyLoadBlob`] holds bytes and an optional
/// MIME type.
///
/// The purpose of having a blob field distinct from the IPC body field is to enable
/// performance optimizations in all sorts of situations. LazyLoadBlobs are only brought
/// across the runtime<>WASM boundary if the process calls `get_blob()`, and this
/// saves lots of work in data-intensive pipelines.
///
/// LazyLoadBlobs also provide a place for less-structured data, such that an IPC body type
/// can be quickly locked in and upgraded within an app-protocol without breaking
/// changes, while still allowing freedom to adjust the contents and shape of a
/// blob. IPC body formats should be rigorously defined.
pub fn blob(mut self, blob: LazyLoadBlob) -> Self {
    self.blob = Some(blob);
    self
}
/// Set the blob's MIME type. If a blob has not been set, it will be set here
/// as an empty vector of bytes. If it has been set, the MIME type will be replaced
/// or created.
pub fn blob_mime(mut self, mime: &str) -> Self {
    if self.blob.is_none() {
        self.blob = Some(LazyLoadBlob {
            mime: Some(mime.to_string()),
            bytes: vec![],
        });
        self
    } else {
        self.blob = Some(LazyLoadBlob {
            mime: Some(mime.to_string()),
            bytes: self.blob.unwrap().bytes,
        });
        self
    }
}
/// Set the blob's bytes. If a blob has not been set, it will be set here with
/// no MIME type. If it has been set, the bytes will be replaced with these bytes.
pub fn blob_bytes<T>(mut self, bytes: T) -> Self
where
    T: Into<Vec<u8>>,
{
    if self.blob.is_none() {
```

```
            self.blob = Some(LazyLoadBlob {
                mime: None,
                bytes: bytes.into(),
            });
            self
        } else {
            self.blob = Some(LazyLoadBlob {
                mime: self.blob.unwrap().mime,
                bytes: bytes.into(),
            });
            self
        }
    }
    /// Set the blob's bytes with a type that implements `TryInto<Vec<u8>>`
    /// and may or may not successfully be set.
    pub fn try_blob_bytes<T>(mut self, bytes: T) -> anyhow::Result<Self>
    where
        T: TryInto<Vec<u8>, Error = anyhow::Error>,
    {
        if self.blob.is_none() {
            self.blob = Some(LazyLoadBlob {
                mime: None,
                bytes: bytes.try_into()?,
            });
            Ok(self)
        } else {
            self.blob = Some(LazyLoadBlob {
                mime: self.blob.unwrap().mime,
                bytes: bytes.try_into()?,
            });
            Ok(self)
        }
    }
    /// Add capabilities to this response. Capabilities are a way to pass
    pub fn capabilities(mut self, capabilities: Vec<Capability>) -> Self {
        self.capabilities = capabilities;
        self
    }
    /// Attempt to send the response. This will only fail if the IPC body field of
    /// the response has not yet been set using `body()` or `try_body()`.
    pub fn send(self) -> anyhow::Result<()> {
        if let Some(body) = self.body {
            crate::send_response(
                &crate::kinode::process::standard::Response {
                    inherit: self.inherit,
                    body,
                    metadata: self.metadata,
                    capabilities: self.capabilities,
                },
                self.blob.as_ref(),
            );
            Ok(())
        } else {
            Err(anyhow::anyhow!("missing IPC body"))
        }
    }
}

impl Default for Response {
    fn default() -> Self {
        Self::new()
    }
}
```

src/lazy_load_blob.rs
==========

```
pub use crate::LazyLoadBlob;

impl std::default::Default for LazyLoadBlob {
    fn default() -> Self {
        LazyLoadBlob {
            mime: None,
            bytes: Vec::new(),
        }
    }
}

impl std::cmp::PartialEq for LazyLoadBlob {
    fn eq(&self, other: &Self) -> bool {
        self.mime == other.mime && self.bytes == other.bytes
    }
}
```

src/message.rs
==========

```
use crate::*;

/// The basic message type. A message is either a request or a response. Best
/// practice when handling a message is to do this:
/// 1. Match on whether it's a request or a response
/// 2. Match on who the message is from (the `source`)
/// 3. Parse and interpret the `body`, `metadata`, and/or `context` based on
/// who the message is from and what your process expects from them.
#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum Message {
    Request {
        source: Address,
        expects_response: Option<u64>,
        body: Vec<u8>,
        metadata: Option<String>,
        capabilities: Vec<Capability>,
    },
    Response {
        source: Address,
        body: Vec<u8>,
        metadata: Option<String>,
        context: Option<Vec<u8>>,
        capabilities: Vec<Capability>,
    },
}

impl Message {
    pub fn is_request(&self) -> bool {
        match self {
            Message::Request { .. } => true,
            Message::Response { .. } => false,
        }
    }
    /// Get the source of a message.
    pub fn source(&self) -> &Address {
        match self {
            Message::Request { source, .. } => source,
            Message::Response { source, .. } => source,
        }
    }
    /// Get the IPC body of a message.
    pub fn body(&self) -> &[u8] {
        match self {
            Message::Request { body, .. } => body,
            Message::Response { body, .. } => body,
        }
```

```
    }
    /// Get the metadata of a message.
    pub fn metadata(&self) -> Option<&str> {
        match self {
            Message::Request { metadata, .. } => metadata.as_ref().map(|s| s.as_str()),
            Message::Response { metadata, .. } => metadata.as_ref().map(|s| s.as_str()),
        }
    }
    /// Get the context of a message.
    pub fn context(&self) -> Option<&[u8]> {
        match self {
            Message::Request { .. } => None,
            Message::Response { context, .. } => context.as_ref().map(|s| s.as_slice()),
        }
    }
    /// Get the blob of a message, if any.
    pub fn blob(&self) -> Option<LazyLoadBlob> {
        crate::get_blob()
    }

    /// Get the capabilities of a message.
    pub fn capabilities(&self) -> &Vec<Capability> {
        match self {
            Message::Request { capabilities, .. } => capabilities,
            Message::Response { capabilities, .. } => capabilities,
        }
    }
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum SendErrorKind {
    Offline,
    Timeout,
}

impl SendErrorKind {
    pub fn is_offline(&self) -> bool {
        matches!(self, SendErrorKind::Offline)
    }
    pub fn is_timeout(&self) -> bool {
        matches!(self, SendErrorKind::Timeout)
    }
}

#[derive(Debug, Clone)]
pub struct SendError {
    pub kind: SendErrorKind,
    pub message: Message,
    pub lazy_load_blob: Option<LazyLoadBlob>,
    pub context: Option<Vec<u8>>,
}

impl SendError {
    pub fn kind(&self) -> &SendErrorKind {
        &self.kind
    }
    pub fn message(&self) -> &Message {
        &self.message
    }
    pub fn blob(&self) -> Option<&LazyLoadBlob> {
        self.lazy_load_blob.as_ref()
    }
    pub fn context(&self) -> Option<&[u8]> {
        self.context.as_deref()
    }
}
```

```
impl std::fmt::Display for SendError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match &self.kind {
            SendErrorKind::Offline => write!(f, "Offline"),
            SendErrorKind::Timeout => write!(f, "Timeout"),
        }
    }
}

impl std::error::Error for SendError {
    fn description(&self) -> &str {
        match &self.kind {
            SendErrorKind::Offline => "Offline",
            SendErrorKind::Timeout => "Timeout",
        }
    }
}

pub fn wit_message_to_message(
    source: Address,
    message: crate::kinode::process::standard::Message,
) -> Message {
    match message {
        crate::kinode::process::standard::Message::Request(req) => Message::Request {
            source,
            expects_response: req.expects_response,
            body: req.body,
            metadata: req.metadata,
            capabilities: req.capabilities,
        },
        crate::kinode::process::standard::Message::Response((resp, context)) => Message::Response {
            source,
            body: resp.body,
            metadata: resp.metadata,
            context,
            capabilities: resp.capabilities,
        },
    }
}
```

src/kv.rs
=========

```
use crate::{get_blob, Message, PackageId, Request};
use serde::{de::DeserializeOwned, Deserialize, Serialize};
use std::marker::PhantomData;
use thiserror::Error;

/// Actions are sent to a specific key value database, "db" is the name,
/// "package_id" is the package. Capabilities are checked, you can access another process's
/// database if it has given you the capability.
#[derive(Debug, Serialize, Deserialize)]
pub struct KvRequest {
    pub package_id: PackageId,
    pub db: String,
    pub action: KvAction,
}

#[derive(Debug, Serialize, Deserialize, Clone)]
pub enum KvAction {
    Open,
    RemoveDb,
    Set { key: Vec<u8>, tx_id: Option<u64> },
    Delete { key: Vec<u8>, tx_id: Option<u64> },
    Get { key: Vec<u8> },
```

```
    BeginTx,
    Commit { tx_id: u64 },
    Backup,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum KvResponse {
    Ok,
    BeginTx { tx_id: u64 },
    Get { key: Vec<u8> },
    Err { error: KvError },
}

#[derive(Debug, Serialize, Deserialize, Error)]
pub enum KvError {
    #[error("kv: DbDoesNotExist")]
    NoDb,
    #[error("kv: KeyNotFound")]
    KeyNotFound,
    #[error("kv: no Tx found")]
    NoTx,
    #[error("kv: No capability: {error}")]
    NoCap { error: String },
    #[error("kv: rocksdb internal error: {error}")]
    RocksDBError { action: String, error: String },
    #[error("kv: input bytes/json/key error: {error}")]
    InputError { error: String },
    #[error("kv: IO error: {error}")]
    IOError { error: String },
}

/// Kv helper struct for a db.
/// Opening or creating a kv will give you a Result<Kv>.
/// You can call it's impl functions to interact with it.
#[derive(Debug, Serialize, Deserialize)]
pub struct Kv<K, V> {
    pub package_id: PackageId,
    pub db: String,
    pub timeout: u64,
    _marker: PhantomData<(K, V)>,
}

impl<K, V> Kv<K, V>
where
    K: Serialize + DeserializeOwned,
    V: Serialize + DeserializeOwned,
{
    /// Get a value.
    pub fn get(&self, key: &K) -> anyhow::Result<V> {
        let key = serde_json::to_vec(key)?;
        let res = Request::new()
            .target(("our", "kv", "distro", "sys"))
            .body(serde_json::to_vec(&KvRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: KvAction::Get { key },
            })?)
            .send_and_await_response(self.timeout)?;

        match res {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<KvResponse>(&body)?;

                match response {
                    KvResponse::Get { .. } => {
                        let bytes = match get_blob() {
```

```
                    Some(bytes) => bytes.bytes,
                    None => return Err(anyhow::anyhow!("kv: no blob")),
                };
                let value = serde_json::from_slice::<V>(&bytes)
                    .map_err(|e| anyhow::anyhow!("Failed to deserialize value: {}", e))?;
                Ok(value)
            }
            KvResponse::Err { error } => Err(error.into()),
            _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
        }
    }
    _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
  }
}

/// Set a value, optionally in a transaction.
pub fn set(&self, key: &K, value: &V, tx_id: Option<u64>) -> anyhow::Result<()> {
  let key = serde_json::to_vec(key)?;
  let value = serde_json::to_vec(value)?;

  let res = Request::new()
    .target(("our", "kv", "distro", "sys"))
    .body(serde_json::to_vec(&KvRequest {
      package_id: self.package_id.clone(),
      db: self.db.clone(),
      action: KvAction::Set { key, tx_id },
    })?)
    .blob_bytes(value)
    .send_and_await_response(self.timeout)?;

  match res {
    Ok(Message::Response { body, .. }) => {
      let response = serde_json::from_slice::<KvResponse>(&body)?;

      match response {
        KvResponse::Ok => Ok(()),
        KvResponse::Err { error } => Err(error.into()),
        _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
      }
    }
    _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
  }
}

/// Delete a value, optionally in a transaction.
pub fn delete(&self, key: &K, tx_id: Option<u64>) -> anyhow::Result<()> {
  let key = serde_json::to_vec(key)?;
  let res = Request::new()
    .target(("our", "kv", "distro", "sys"))
    .body(serde_json::to_vec(&KvRequest {
      package_id: self.package_id.clone(),
      db: self.db.clone(),
      action: KvAction::Delete { key, tx_id },
    })?)
    .send_and_await_response(self.timeout)?;

  match res {
    Ok(Message::Response { body, .. }) => {
      let response = serde_json::from_slice::<KvResponse>(&body)?;

      match response {
        KvResponse::Ok => Ok(()),
        KvResponse::Err { error } => Err(error.into()),
        _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
      }
    }
```

```
            _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
        }
    }

    /// Begin a transaction.
    pub fn begin_tx(&self) -> anyhow::Result<u64> {
        let res = Request::new()
            .target(("our", "kv", "distro", "sys"))
            .body(serde_json::to_vec(&KvRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: KvAction::BeginTx,
            })?)
            .send_and_await_response(self.timeout)?;

        match res {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<KvResponse>(&body)?;

                match response {
                    KvResponse::BeginTx { tx_id } => Ok(tx_id),
                    KvResponse::Err { error } => Err(error.into()),
                    _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
                }
            }
            _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
        }
    }

    /// Commit a transaction.
    pub fn commit_tx(&self, tx_id: u64) -> anyhow::Result<()> {
        let res = Request::new()
            .target(("our", "kv", "distro", "sys"))
            .body(serde_json::to_vec(&KvRequest {
                package_id: self.package_id.clone(),
                db: self.db.clone(),
                action: KvAction::Commit { tx_id },
            })?)
            .send_and_await_response(self.timeout)?;

        match res {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<KvResponse>(&body)?;

                match response {
                    KvResponse::Ok => Ok(()),
                    KvResponse::Err { error } => Err(error.into()),
                    _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
                }
            }
            _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
        }
    }
}

/// Opens or creates a kv db.
pub fn open<K, V>(package_id: PackageId, db: &str, timeout: Option<u64>) -> anyhow::Result<Kv<K, V>
where
    K: Serialize + DeserializeOwned,
    V: Serialize + DeserializeOwned,
{
    let timeout = timeout.unwrap_or(5);

    let res = Request::new()
        .target(("our", "kv", "distro", "sys"))
        .body(serde_json::to_vec(&KvRequest {
```

```
            package_id: package_id.clone(),
            db: db.to_string(),
            action: KvAction::Open,
        })?)
        .send_and_await_response(timeout)?;

    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<KvResponse>(&body)?;

            match response {
                KvResponse::Ok => Ok(Kv {
                    package_id,
                    db: db.to_string(),
                    timeout,
                    _marker: PhantomData,
                }),
                KvResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
    }
}

/// Removes and deletes a kv db.
pub fn remove_db(package_id: PackageId, db: &str, timeout: Option<u64>) -> anyhow::Result<()> {
    let timeout = timeout.unwrap_or(5);

    let res = Request::new()
        .target(("our", "kv", "distro", "sys"))
        .body(serde_json::to_vec(&KvRequest {
            package_id: package_id.clone(),
            db: db.to_string(),
            action: KvAction::RemoveDb,
        })?)
        .send_and_await_response(timeout)?;

    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<KvResponse>(&body)?;

            match response {
                KvResponse::Ok => Ok(()),
                KvResponse::Err { error } => Err(error.into()),
                _ => Err(anyhow::anyhow!("kv: unexpected response {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("kv: unexpected message: {:?}", res)),
    }
}
```

src/package_id.rs
==========

```
use crate::ProcessIdParseError;
use serde::{Deserialize, Serialize};
use std::hash::Hash;

/// PackageId is like a ProcessId, but for a package. Only contains the name
/// of the package and the name of the publisher.
#[derive(Hash, Eq, PartialEq, Debug, Clone, Serialize, Deserialize)]
pub struct PackageId {
    package_name: String,
    publisher_node: String,
}
```

```
impl PackageId {
    /// Create a new `PackageId`.
    pub fn new(package_name: &str, publisher_node: &str) -> Self {
        PackageId {
            package_name: package_name.into(),
            publisher_node: publisher_node.into(),
        }
    }
    /// Read the package name from a `PackageId`.
    pub fn package(&self) -> &str {
        &self.package_name
    }
    /// Read the publisher node ID from a `PackageId`. Note that `PackageId`
    /// segments are not parsed for validity, and a node ID stored here is
    /// not guaranteed to be a valid ID in the name system, or be connected
    /// to an identity at all.
    pub fn publisher(&self) -> &str {
        &self.publisher_node
    }
}

impl std::str::FromStr for PackageId {
    type Err = ProcessIdParseError;
    /// Attempt to parse a `PackageId` from a string. The string must
    /// contain exactly two segments, where segments are strings separated
    /// by a colon `:`. The segments cannot themselves contain colons.
    /// Please note that while any string without colons will parse successfully
    /// to create a `PackageId`, not all strings without colons are actually
    /// valid usernames, which the `publisher_node` field of a `PackageId` will
    /// always in practice be.
    fn from_str(input: &str) -> Result<Self, Self::Err> {
        // split string on colons into 2 segments
        let mut segments = input.split(':');
        let package_name = segments
            .next()
            .ok_or(ProcessIdParseError::MissingField)?
            .to_string();
        let publisher_node = segments
            .next()
            .ok_or(ProcessIdParseError::MissingField)?
            .to_string();
        if segments.next().is_some() {
            return Err(ProcessIdParseError::TooManyColons);
        }
        Ok(PackageId {
            package_name,
            publisher_node,
        })
    }
}

impl std::fmt::Display for PackageId {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}:{}", self.package_name, self.publisher_node)
    }
}
```

src/lib.rs
==========

```
//! kinode process standard library for Rust compiled to WASM
//! Must be used in context of bindings generated by `kinode.wit`.
//!
//! This library provides a set of functions for interacting with the kinode
//! kernel interface, which is a WIT file. The types generated by this file
```

```
//! are available in processes via the wit_bindgen macro, if a process needs
//! to use them directly. However, the most convenient way to do most things
//! will be via this library.
//!
//! We define wrappers over the wit bindings to make them easier to use.
//! This library encourages the use of IPC body and metadata types serialized and
//! deserialized to JSON, which is not optimal for performance, but useful
//! for applications that want to maximize composability and introspectability.
//! For blobs, we recommend bincode to serialize and deserialize to bytes.
//!
pub use crate::kinode::process::standard::*;
use serde::{Deserialize, Serialize};
use serde_json::Value;

wit_bindgen::generate!({
    path: "kinode-wit",
    world: "lib",
});

/// Interact with the eth provider module.
pub mod eth;
/// Interact with the HTTP server and client modules.
/// Contains types from the `http` crate to use as well.
pub mod http;
/// The types that the kernel itself uses -- warning -- these will
/// be incompatible with WIT types in some cases, leading to annoying errors.
/// Use only to interact with the kernel or runtime in certain ways.
pub mod kernel_types;
/// Interact with the key_value module
pub mod kv;
/// Interact with the networking module
/// For configuration, debugging, and creating signatures with networking key.
pub mod net;
/// Interact with the sqlite module
pub mod sqlite;
/// Interact with the timer runtime module.
pub mod timer;
/// Interact with the virtual filesystem
pub mod vfs;

// Types

mod package_id;
pub use package_id::PackageId;
mod process_id;
pub use process_id::{ProcessId, ProcessIdParseError};
mod address;
pub use address::{Address, AddressParseError};
mod request;
pub use request::Request;
mod response;
pub use response::Response;
mod message;
use message::wit_message_to_message;
pub use message::{Message, SendError, SendErrorKind};
mod on_exit;
pub use on_exit::OnExit;
mod capability;
pub use capability::Capability;
mod lazy_load_blob;
pub use lazy_load_blob::LazyLoadBlob;

/// Implement the wit-bindgen specific code that the kernel uses to hook into
/// a process. Write an `init(our: Address)` function and call it with this.
#[macro_export]
macro_rules! call_init {
```

```
    ($init_func:ident) => {
        struct Component;
        impl Guest for Component {
            fn init(our: String) {
                let our: Address = our.parse().unwrap();
                $init_func(our);
            }
        }
        export!(Component);
    };
}

/// Override the println! macro to print to the terminal. Uses the
/// `print_to_terminal` function from the WIT interface on maximally-verbose
/// mode, i.e., this print will always show up in the terminal. To control
/// the verbosity, use the `print_to_terminal` function directly.
#[macro_export]
macro_rules! println {
    () => {
        $crate::print_to_terminal(0, "\n");
    };
    ($($arg:tt)*) => {{
        $crate::print_to_terminal(0, &format!($($arg)*));
    }};
}

/// Await the next message sent to this process. The runtime will handle the
/// queueing of incoming messages, and calling this function will provide the next one.
/// Interwoven with incoming messages are errors from the network. If your process
/// attempts to send a message to another node, that message may bounce back with
/// a `SendError`. Those should be handled here.
///
/// TODO: example of usage
pub fn await_message() -> Result<Message, SendError> {
    match crate::receive() {
        Ok((source, message)) => Ok(wit_message_to_message(source, message)),
        Err((send_err, context)) => Err(SendError {
            kind: match send_err.kind {
                crate::kinode::process::standard::SendErrorKind::Offline => SendErrorKind::Offline,
                crate::kinode::process::standard::SendErrorKind::Timeout => SendErrorKind::Timeout,
            },
            message: wit_message_to_message(
                Address::new("our", ProcessId::new(Some("net"), "distro", "sys")),
                send_err.message,
            ),
            lazy_load_blob: send_err.lazy_load_blob,
            context,
        }),
    }
}

/// Simple wrapper over spawn() in WIT to make use of our good types
pub fn spawn(
    name: Option<&str>,
    wasm_path: &str,
    on_exit: OnExit,
    request_capabilities: Vec<Capability>,
    grant_capabilities: Vec<ProcessId>,
    public: bool,
) -> Result<ProcessId, SpawnError> {
    crate::kinode::process::standard::spawn(
        name,
        wasm_path,
        &on_exit._to_standard().map_err(|_e| SpawnError::NameTaken)?,
        &request_capabilities,
        &grant_capabilities,
```

```
        public,
    )
}

/// Create a blob with no MIME type and a generic type, plus a serializer
/// function that turns that type into bytes.
///
/// Example: TODO
pub fn make_blob<T, F>(blob: &T, serializer: F) -> anyhow::Result<LazyLoadBlob>
where
    F: Fn(&T) -> anyhow::Result<Vec<u8>>,
{
    Ok(LazyLoadBlob {
        mime: None,
        bytes: serializer(blob)?,
    })
}

/// Fetch the blob of the most recent message we've received. Returns `None`
/// if that message had no blob. If it does have one, attempt to deserialize
/// it from bytes with the provided function.
///
/// Example:
/// ```
/// get_typed_blob(|bytes| Ok(bincode::deserialize(bytes)?)).unwrap_or(MyType {
///     field: HashMap::new(),
///     field_two: HashSet::new(),
/// });
/// ```
pub fn get_typed_blob<T, F>(deserializer: F) -> Option<T>
where
    F: Fn(&[u8]) -> anyhow::Result<T>,
{
    match crate::get_blob() {
        Some(blob) => match deserializer(&blob.bytes) {
            Ok(thing) => Some(thing),
            Err(_) => None,
        },
        None => None,
    }
}

/// Fetch the persisted state blob associated with this process. This blob is saved
/// using the [`set_state`] function. Returns `None` if this process has no saved state.
/// If it does, attempt to deserialize it from bytes with the provided function.
///
/// Example:
/// ```
/// get_typed_state(|bytes| Ok(bincode::deserialize(bytes)?)).unwrap_or(MyStateType {
///     field: HashMap::new(),
///     field_two: HashSet::new(),
/// });
/// ```
pub fn get_typed_state<T, F>(deserializer: F) -> Option<T>
where
    F: Fn(&[u8]) -> anyhow::Result<T>,
{
    match crate::get_state() {
        Some(bytes) => match deserializer(&bytes) {
            Ok(thing) => Some(thing),
            Err(_) => None,
        },
        None => None,
    }
}
```

```
/// See if we have the capability to message a certain process.
/// Note if you have not saved the capability, you will not be able to message the other process.
pub fn can_message(address: &Address) -> bool {
    crate::our_capabilities()
        .iter()
        .any(|cap| cap.params == "\"messaging\"" && cap.issuer == *address)
}

/// Get a capability in our store
pub fn get_capability(our: &Address, params: &str) -> Option<Capability> {
    let params = serde_json::from_str::<Value>(params).unwrap_or_default();
    crate::our_capabilities()
        .iter()
        .find(|cap| {
            let cap_params = serde_json::from_str::<Value>(&cap.params).unwrap_or_default();
            cap.issuer == *our && params == cap_params
        })
        .cloned()
}

/// get the next message body from the message queue, or propagate the error
pub fn await_next_message_body() -> Result<Vec<u8>, SendError> {
    match await_message() {
        Ok(msg) => Ok(msg.body().to_vec()),
        Err(e) => Err(e.into()),
    }
}
```

src/address.rs
==========

```
pub use crate::{Address, PackageId, ProcessId};
use serde::{Deserialize, Serialize};
use std::hash::{Hash, Hasher};

/// Address is defined in the wit bindings, but constructors and methods here.
/// An `Address` is a combination of a node ID (string) and a [`ProcessId`]. It is
/// used in the Request/Response pattern to indicate which process on a given node
/// in the network to direct the message to. The formatting structure for
/// an Address is `node@process_name:package_name:publisher_node`
impl Address {
    /// Create a new `Address`. Takes a node ID and a process ID.
    pub fn new<T, U>(node: T, process: U) -> Address
    where
        T: Into<String>,
        U: Into<ProcessId>,
    {
        Address {
            node: node.into(),
            process: process.into(),
        }
    }
    /// Read the node ID from an `Address`.
    pub fn node(&self) -> &str {
        &self.node
    }
    /// Read the process name from an `Address`.
    pub fn process(&self) -> &str {
        &self.process.process_name
    }
    /// Read the package name from an `Address`.
    pub fn package(&self) -> &str {
        &self.process.package_name
    }
    /// Read the publisher node ID from an `Address`. Note that `Address`
    /// segments are not parsed for validity, and a node ID stored here is
```

```
    /// not guaranteed to be a valid ID in the name system, or be connected
    /// to an identity at all.
    pub fn publisher(&self) -> &str {
        &self.process.publisher_node
    }
    /// Read the package_id (package + publisher) from an `Address`.
    pub fn package_id(&self) -> PackageId {
        PackageId::new(self.package(), self.publisher())
    }
}

impl std::str::FromStr for Address {
    type Err = AddressParseError;
    /// Attempt to parse an `Address` from a string. The formatting structure for
    /// an Address is `node@process_name:package_name:publisher_node`.
    ///
    /// TODO: clarify if `@` can be present in process name / package name / publisher name
    ///
    /// TODO: ensure `:` cannot sneak into first segment
    fn from_str(input: &str) -> Result<Self, AddressParseError> {
        // split string on colons into 4 segments,
        // first one with @, next 3 with :
        let mut name_rest = input.split('@');
        let node = name_rest
            .next()
            .ok_or(AddressParseError::MissingField)?
            .to_string();
        let mut segments = name_rest
            .next()
            .ok_or(AddressParseError::MissingNodeId)?
            .split(':');
        let process_name = segments
            .next()
            .ok_or(AddressParseError::MissingField)?
            .to_string();
        let package_name = segments
            .next()
            .ok_or(AddressParseError::MissingField)?
            .to_string();
        let publisher_node = segments
            .next()
            .ok_or(AddressParseError::MissingField)?
            .to_string();
        if segments.next().is_some() {
            return Err(AddressParseError::TooManyColons);
        }
        Ok(Address {
            node,
            process: ProcessId {
                process_name,
                package_name,
                publisher_node,
            },
        })
    }
}

impl Serialize for Address {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::ser::Serializer,
    {
        format!("{}", self).serialize(serializer)
    }
}
```

```
impl<'a> Deserialize<'a> for Address {
    fn deserialize<D>(deserializer: D) -> Result<Address, D::Error>
    where
        D: serde::de::Deserializer<'a>,
    {
        let s = String::deserialize(deserializer)?;
        s.parse().map_err(serde::de::Error::custom)
    }
}

impl Hash for Address {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.node.hash(state);
        self.process.hash(state);
    }
}

impl Eq for Address {}

impl PartialEq for Address {
    fn eq(&self, other: &Self) -> bool {
        self.node == other.node && self.process == other.process
    }
}

impl From<&Address> for Address {
    fn from(input: &Address) -> Self {
        input.clone()
    }
}

impl<T, U, V, W> From<(T, U, V, W)> for Address
where
    T: Into<String>,
    U: Into<&'static str>,
    V: Into<&'static str>,
    W: Into<&'static str>,
{
    fn from(input: (T, U, V, W)) -> Self {
        Address::new(
            input.0.into(),
            (input.1.into(), input.2.into(), input.3.into()),
        )
    }
}

impl<T> From<(&str, T)> for Address
where
    T: Into<ProcessId>,
{
    fn from(input: (&str, T)) -> Self {
        Address::new(input.0, input.1)
    }
}

impl std::fmt::Display for Address {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}@{}", self.node, self.process)
    }
}

/// Error type for parsing an `Address` from a string.
#[derive(Debug)]
pub enum AddressParseError {
    TooManyColons,
    MissingNodeId,
```

```
    MissingField,
}

impl std::fmt::Display for AddressParseError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{}",
            match self {
                AddressParseError::TooManyColons => "Too many colons in ProcessId string",
                AddressParseError::MissingNodeId => "Node ID missing",
                AddressParseError::MissingField => "Missing field in ProcessId string",
            }
        )
    }
}

impl std::error::Error for AddressParseError {
    fn description(&self) -> &str {
        match self {
            AddressParseError::TooManyColons => "Too many colons in ProcessId string",
            AddressParseError::MissingNodeId => "Node ID missing",
            AddressParseError::MissingField => "Missing field in ProcessId string",
        }
    }
}

src/eth.rs
==========

use crate::{Message, Request as KiRequest};
pub use alloy_primitives::{Address, BlockHash, BlockNumber, Bytes, TxHash, U128, U256, U64, U8};
pub use alloy_rpc_types::pubsub::{Params, SubscriptionKind, SubscriptionResult};
pub use alloy_rpc_types::{
    request::{TransactionInput, TransactionRequest},
    Block, BlockId, BlockNumberOrTag, FeeHistory, Filter, FilterBlockOption, Log, Transaction,
    TransactionReceipt,
};
use serde::{Deserialize, Serialize};
use std::collections::{HashMap, HashSet};

//
//  types mirrored from runtime module
//

/// The Action and Request type that can be made to eth:distro:sys. Any process with messaging
/// capabilities can send this action to the eth provider.
///
/// Will be serialized and deserialized using `serde_json::to_vec` and `serde_json::from_slice`.
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum EthAction {
    /// Subscribe to logs with a custom filter. ID is to be used to unsubscribe.
    /// Logs come in as alloy_rpc_types::pubsub::SubscriptionResults
    SubscribeLogs {
        sub_id: u64,
        chain_id: u64,
        kind: SubscriptionKind,
        params: Params,
    },
    /// Kill a SubscribeLogs subscription of a given ID, to stop getting updates.
    UnsubscribeLogs(u64),
    /// Raw request. Used by kinode_process_lib.
    Request {
        chain_id: u64,
        method: String,
        params: serde_json::Value,
```

```
    },
}

/// Incoming `Request` containing subscription updates or errors that processes will receive.
/// Can deserialize all incoming requests from eth:distro:sys to this type.
///
/// Will be serialized and deserialized using `serde_json::to_vec` and `serde_json::from_slice`.
pub type EthSubResult = Result<EthSub, EthSubError>;

/// Incoming type for successful subscription updates.
#[derive(Debug, Serialize, Deserialize)]
pub struct EthSub {
    pub id: u64,
    pub result: SubscriptionResult,
}

/// If your subscription is closed unexpectedly, you will receive this.
#[derive(Debug, Serialize, Deserialize)]
pub struct EthSubError {
    pub id: u64,
    pub error: String,
}

/// The Response type which a process will get from requesting with an [`EthAction`] will be
/// of this type, serialized and deserialized using `serde_json::to_vec`
/// and `serde_json::from_slice`.
///
/// In the case of an [`EthAction::SubscribeLogs`] request, the response will indicate if
/// the subscription was successfully created or not.
#[derive(Debug, Serialize, Deserialize)]
pub enum EthResponse {
    Ok,
    Response { value: serde_json::Value },
    Err(EthError),
}

#[derive(Debug, Serialize, Deserialize, PartialEq)]
pub enum EthError {
    /// provider module cannot parse message
    MalformedRequest,
    /// No RPC provider for the chain
    NoRpcForChain,
    /// Subscription closed
    SubscriptionClosed(u64),
    /// Invalid method
    InvalidMethod(String),
    /// Invalid parameters
    InvalidParams,
    /// Permission denied
    PermissionDenied,
    /// RPC timed out
    RpcTimeout,
    /// RPC gave garbage back
    RpcMalformedResponse,
}

/// The action type used for configuring eth:distro:sys. Only processes which have the "root"
/// capability from eth:distro:sys can successfully send this action.
///
/// NOTE: changes to config will not be persisted between boots, they must be saved in .env
/// to be reflected between boots. TODO: can change this
#[derive(Debug, Serialize, Deserialize)]
pub enum EthConfigAction {
    /// Add a new provider to the list of providers.
    AddProvider(ProviderConfig),
    /// Remove a provider from the list of providers.
```

```
    /// The tuple is (chain_id, node_id/rpc_url).
    RemoveProvider((u64, String)),
    /// make our provider public
    SetPublic,
    /// make our provider not-public
    SetPrivate,
    /// add node to whitelist on a provider
    AllowNode(String),
    /// remove node from whitelist on a provider
    UnallowNode(String),
    /// add node to blacklist on a provider
    DenyNode(String),
    /// remove node from blacklist on a provider
    UndenyNode(String),
    /// Set the list of providers to a new list.
    /// Replaces all existing saved provider configs.
    SetProviders(SavedConfigs),
    /// Get the list of current providers as a [`SavedConfigs`] object.
    GetProviders,
    /// Get the current access settings.
    GetAccessSettings,
    /// Get the state of calls and subscriptions. Used for debugging.
    GetState,
}

/// Response type from an [`EthConfigAction`] request.
#[derive(Debug, Serialize, Deserialize)]
pub enum EthConfigResponse {
    Ok,
    /// Response from a GetProviders request.
    /// Note the [`crate::kernel_types::KnsUpdate`] will only have the correct `name` field.
    /// The rest of the Update is not saved in this module.
    Providers(SavedConfigs),
    /// Response from a GetAccessSettings request.
    AccessSettings(AccessSettings),
    /// Permission denied due to missing capability
    PermissionDenied,
    /// Response from a GetState request
    State {
        active_subscriptions: HashMap<crate::Address, HashMap<u64, Option<String>>>, // None if local, S
        outstanding_requests: HashSet<u64>,
    },
}

/// Settings for our ETH provider
#[derive(Clone, Debug, Deserialize, Serialize)]
pub struct AccessSettings {
    pub public: bool,          // whether or not other nodes can access through us
    pub allow: HashSet<String>, // whitelist for access (only used if public == false)
    pub deny: HashSet<String>,  // blacklist for access (always used)
}

pub type SavedConfigs = Vec<ProviderConfig>;

/// Provider config. Can currently be a node or a ws provider instance.
#[derive(Clone, Debug, Deserialize, Serialize)]
pub struct ProviderConfig {
    pub chain_id: u64,
    pub trusted: bool,
    pub provider: NodeOrRpcUrl,
}

#[derive(Clone, Debug, Deserialize, Serialize)]
pub enum NodeOrRpcUrl {
    Node {
        kns_update: crate::kernel_types::KnsUpdate,
```

```
        use_as_provider: bool, // for routers inside saved config
    },
    RpcUrl(String),
}

impl std::cmp::PartialEq<str> for NodeOrRpcUrl {
    fn eq(&self, other: &str) -> bool {
        match self {
            NodeOrRpcUrl::Node { kns_update, .. } => kns_update.name == other,
            NodeOrRpcUrl::RpcUrl(url) => url == other,
        }
    }
}

/// An EVM chain provider. Create this object to start making RPC calls.
/// Set the chain_id to determine which chain to call: requests will fail
/// unless the node this process is running on has access to a provider
/// for that chain.
pub struct Provider {
    chain_id: u64,
    request_timeout: u64,
}

impl Provider {
    /// Instantiate a new provider.
    pub fn new(chain_id: u64, request_timeout: u64) -> Self {
        Self {
            chain_id,
            request_timeout,
        }
    }
    /// Sends a request based on the specified `EthAction` and parses the response.
    ///
    /// This function constructs a request targeting the Ethereum distribution system, serializes the provide
    /// and sends it. It awaits a response with a specified timeout, then attempts to parse the response into
    /// type `T`. This method is generic and can be used for various Ethereum actions by specifying the app
    /// and return type `T`.
    pub fn send_request_and_parse_response<T: serde::de::DeserializeOwned>(
        &self,
        action: EthAction,
    ) -> Result<T, EthError> {
        let resp = KiRequest::new()
            .target(("our", "eth", "distro", "sys"))
            .body(serde_json::to_vec(&action).unwrap())
            .send_and_await_response(self.request_timeout)
            .unwrap()
            .map_err(|_| EthError::RpcTimeout)?;

        match resp {
            Message::Response { body, .. } => match serde_json::from_slice::<EthResponse>(&body) {
                Ok(EthResponse::Response { value }) => {
                    serde_json::from_value::<T>(value).map_err(|_| EthError::RpcMalformedResponse)
                }
                Ok(EthResponse::Err(e)) => Err(e),
                _ => Err(EthError::RpcMalformedResponse),
            },
            _ => Err(EthError::RpcMalformedResponse),
        }
    }

    /// Retrieves the current block number.
    ///
    /// # Returns
    /// A `Result<u64, EthError>` representing the current block number.
    pub fn get_block_number(&self) -> Result<u64, EthError> {
        let action = EthAction::Request {
```

```
        chain_id: self.chain_id,
        method: "eth_blockNumber".to_string(),
        params: ().into(),
    };

    let res = self.send_request_and_parse_response::<U64>(action)?;
    Ok(res.to::<u64>())
}

/// Retrieves the balance of the given address at the specified block.
///
/// # Parameters
/// - `address`: The address to query the balance for.
/// - `tag`: Optional block ID to specify the block at which the balance is queried.
///
/// # Returns
/// A `Result<U256, EthError>` representing the balance of the address.
pub fn get_balance(&self, address: Address, tag: Option<BlockId>) -> Result<U256, EthError> {
    let params = serde_json::to_value((
        address,
        tag.unwrap_or(BlockId::Number(BlockNumberOrTag::Latest)),
    ))
    .unwrap();
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getBalance".to_string(),
        params,
    };

    self.send_request_and_parse_response::<U256>(action)
}

/// Retrieves logs based on a filter.
///
/// # Parameters
/// - `filter`: The filter criteria for the logs.
///
/// # Returns
/// A `Result<Vec<Log>, EthError>` containing the logs that match the filter.
pub fn get_logs(&self, filter: &Filter) -> Result<Vec<Log>, EthError> {
    // NOTE: filter must be encased by a tuple to be serialized correctly
    let Ok(params) = serde_json::to_value((filter,)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getLogs".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Vec<Log>>(action)
}

/// Retrieves the current gas price.
///
/// # Returns
/// A `Result<U256, EthError>` representing the current gas price.
pub fn get_gas_price(&self) -> Result<U256, EthError> {
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_gasPrice".to_string(),
        params: ().into(),
    };

    self.send_request_and_parse_response::<U256>(action)
}
```

```
/// Retrieves the number of transactions sent from the given address.
///
/// # Parameters
/// - `address`: The address to query the transaction count for.
/// - `tag`: Optional block ID to specify the block at which the count is queried.
///
/// # Returns
/// A `Result<U256, EthError>` representing the number of transactions sent from the address.
pub fn get_transaction_count(
    &self,
    address: Address,
    tag: Option<BlockId>,
) -> Result<U256, EthError> {
    let Ok(params) = serde_json::to_value((address, tag.unwrap_or_default())) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getTransactionCount".to_string(),
        params,
    };

    self.send_request_and_parse_response::<U256>(action)
}

/// Retrieves a block by its hash.
///
/// # Parameters
/// - `hash`: The hash of the block to retrieve.
/// - `full_tx`: Whether to return full transaction objects or just their hashes.
///
/// # Returns
/// A `Result<Option<Block>, EthError>` representing the block, if found.
pub fn get_block_by_hash(
    &self,
    hash: BlockHash,
    full_tx: bool,
) -> Result<Option<Block>, EthError> {
    let Ok(params) = serde_json::to_value((hash, full_tx)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getBlockByHash".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Option<Block>>(action)
}
/// Retrieves a block by its number or tag.
///
/// # Parameters
/// - `number`: The number or tag of the block to retrieve.
/// - `full_tx`: Whether to return full transaction objects or just their hashes.
///
/// # Returns
/// A `Result<Option<Block>, EthError>` representing the block, if found.
pub fn get_block_by_number(
    &self,
    number: BlockNumberOrTag,
    full_tx: bool,
) -> Result<Option<Block>, EthError> {
    let Ok(params) = serde_json::to_value((number, full_tx)) else {
        return Err(EthError::InvalidParams);
    };
```

```
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getBlockByNumber".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Option<Block>>(action)
}

/// Retrieves the storage at a given address and key.
///
/// # Parameters
/// - `address`: The address of the storage to query.
/// - `key`: The key of the storage slot to retrieve.
/// - `tag`: Optional block ID to specify the block at which the storage is queried.
///
/// # Returns
/// A `Result<Bytes, EthError>` representing the data stored at the given address and key.
pub fn get_storage_at(
    &self,
    address: Address,
    key: U256,
    tag: Option<BlockId>,
) -> Result<Bytes, EthError> {
    let Ok(params) = serde_json::to_value((address, key, tag.unwrap_or_default())) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getStorageAt".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Bytes>(action)
}

/// Retrieves the code at a given address.
///
/// # Parameters
/// - `address`: The address of the code to query.
/// - `tag`: The block ID to specify the block at which the code is queried.
///
/// # Returns
/// A `Result<Bytes, EthError>` representing the code stored at the given address.
pub fn get_code_at(&self, address: Address, tag: BlockId) -> Result<Bytes, EthError> {
    let Ok(params) = serde_json::to_value((address, tag)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getCode".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Bytes>(action)
}

/// Retrieves a transaction by its hash.
///
/// # Parameters
/// - `hash`: The hash of the transaction to retrieve.
///
/// # Returns
/// A `Result<Option<Transaction>, EthError>` representing the transaction, if found.
pub fn get_transaction_by_hash(&self, hash: TxHash) -> Result<Option<Transaction>, EthError> {
    // NOTE: hash must be encased by a tuple to be serialized correctly
```

```
    let Ok(params) = serde_json::to_value((hash,)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getTransactionByHash".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Option<Transaction>>(action)
}

/// Retrieves the receipt of a transaction by its hash.
///
/// # Parameters
/// - `hash`: The hash of the transaction for which the receipt is requested.
///
/// # Returns
/// A `Result<Option<TransactionReceipt>, EthError>` representing the transaction receipt, if found.
pub fn get_transaction_receipt(
    &self,
    hash: TxHash,
) -> Result<Option<TransactionReceipt>, EthError> {
    // NOTE: hash must be encased by a tuple to be serialized correctly
    let Ok(params) = serde_json::to_value((hash,)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_getTransactionReceipt".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Option<TransactionReceipt>>(action)
}

/// Estimates the amount of gas that a transaction will consume.
///
/// # Parameters
/// - `tx`: The transaction request object containing the details of the transaction to estimate gas for.
/// - `block`: Optional block ID to specify the block at which the gas estimate should be made.
///
/// # Returns
/// A `Result<U256, EthError>` representing the estimated gas amount.
pub fn estimate_gas(
    &self,
    tx: TransactionRequest,
    block: Option<BlockId>,
) -> Result<U256, EthError> {
    let Ok(params) = serde_json::to_value((tx, block.unwrap_or_default())) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_estimateGas".to_string(),
        params,
    };

    self.send_request_and_parse_response::<U256>(action)
}

/// Retrieves the list of accounts controlled by the node.
///
/// # Returns
/// A `Result<Vec<Address>, EthError>` representing the list of accounts.
/// Note: This function may return an empty list depending on the node's configuration and capabilities.
```

```
pub fn get_accounts(&self) -> Result<Vec<Address>, EthError> {
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_accounts".to_string(),
        params: serde_json::Value::Array(vec![]),
    };

    self.send_request_and_parse_response::<Vec<Address>>(action)
}

/// Retrieves the fee history for a given range of blocks.
///
/// # Parameters
/// - `block_count`: The number of blocks to include in the history.
/// - `last_block`: The ending block number or tag for the history range.
/// - `reward_percentiles`: A list of percentiles to report fee rewards for.
///
/// # Returns
/// A `Result<FeeHistory, EthError>` representing the fee history for the specified range.
pub fn get_fee_history(
    &self,
    block_count: U256,
    last_block: BlockNumberOrTag,
    reward_percentiles: Vec<f64>,
) -> Result<FeeHistory, EthError> {
    let Ok(params) = serde_json::to_value((block_count, last_block, reward_percentiles)) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_feeHistory".to_string(),
        params,
    };

    self.send_request_and_parse_response::<FeeHistory>(action)
}

/// Executes a call transaction, which is directly executed in the VM of the node, but never mined into th
///
/// # Parameters
/// - `tx`: The transaction request object containing the details of the call.
/// - `block`: Optional block ID to specify the block at which the call should be made.
///
/// # Returns
/// A `Result<Bytes, EthError>` representing the result of the call.
pub fn call(&self, tx: TransactionRequest, block: Option<BlockId>) -> Result<Bytes, EthError> {
    let Ok(params) = serde_json::to_value((tx, block.unwrap_or_default())) else {
        return Err(EthError::InvalidParams);
    };
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_call".to_string(),
        params,
    };

    self.send_request_and_parse_response::<Bytes>(action)
}

/// Sends a raw transaction to the network.
///
/// # Parameters
/// - `tx`: The raw transaction data.
///
/// # Returns
/// A `Result<TxHash, EthError>` representing the hash of the transaction once it has been sent.
pub fn send_raw_transaction(&self, tx: Bytes) -> Result<TxHash, EthError> {
```

```
    let action = EthAction::Request {
        chain_id: self.chain_id,
        method: "eth_sendRawTransaction".to_string(),
        // NOTE: tx must be encased by a tuple to be serialized correctly
        params: serde_json::to_value((tx,)).unwrap(),
    };

    self.send_request_and_parse_response::<TxHash>(action)
}

/// Subscribes to logs without waiting for a confirmation.
///
/// # Parameters
/// - `sub_id`: The subscription ID to be used for unsubscribing.
/// - `filter`: The filter criteria for the logs.
///
/// # Returns
/// A `Result<(), EthError>` indicating whether the subscription was created.
pub fn subscribe(&self, sub_id: u64, filter: Filter) -> Result<(), EthError> {
    let action = EthAction::SubscribeLogs {
        sub_id,
        chain_id: self.chain_id,
        kind: SubscriptionKind::Logs,
        params: Params::Logs(Box::new(filter)),
    };

    let Ok(body) = serde_json::to_vec(&action) else {
        return Err(EthError::InvalidParams);
    };

    let resp = KiRequest::new()
        .target(("our", "eth", "distro", "sys"))
        .body(body)
        .send_and_await_response(self.request_timeout)
        .unwrap()
        .map_err(|_| EthError::RpcTimeout)?;

    match resp {
        Message::Response { body, .. } => {
            let response = serde_json::from_slice::<EthResponse>(&body);
            match response {
                Ok(EthResponse::Ok) => Ok(()),
                Ok(EthResponse::Err(e)) => Err(e),
                _ => Err(EthError::RpcMalformedResponse),
            }
        }
        _ => Err(EthError::RpcMalformedResponse),
    }
}

/// Unsubscribes from a previously created subscription.
///
/// # Parameters
/// - `sub_id`: The subscription ID to unsubscribe from.
///
/// # Returns
/// A `Result<(), EthError>` indicating whether the subscription was cancelled.
pub fn unsubscribe(&self, sub_id: u64) -> Result<(), EthError> {
    let action = EthAction::UnsubscribeLogs(sub_id);

    let resp = KiRequest::new()
        .target(("our", "eth", "distro", "sys"))
        .body(serde_json::to_vec(&action).map_err(|_| EthError::MalformedRequest)?)
        .send_and_await_response(self.request_timeout)
        .unwrap()
        .map_err(|_| EthError::RpcTimeout)?;
```

```
        match resp {
            Message::Response { body, .. } => match serde_json::from_slice::<EthResponse>(&body) {
                Ok(EthResponse::Ok) => Ok(()),
                _ => Err(EthError::RpcMalformedResponse),
            },
            _ => Err(EthError::RpcMalformedResponse),
        }
    }
}
```

src/request.rs
==============

```
use crate::*;

/// Request builder. Use [`Request::new()`] to start a request, then build it,
/// then call [`Request::send()`] on it to fire.
#[derive(Clone, Debug)]
pub struct Request {
    pub target: Option<Address>,
    pub inherit: bool,
    pub timeout: Option<u64>,
    pub body: Option<Vec<u8>>,
    pub metadata: Option<String>,
    pub blob: Option<LazyLoadBlob>,
    pub context: Option<Vec<u8>>,
    pub capabilities: Vec<Capability>,
}

#[allow(dead_code)]
impl Request {
    /// Start building a new `Request`. In order to successfully send, a
    /// `Request` must have at least a `target` and an `body`. Calling send
    /// on this before filling out these fields will result in an error.
    pub fn new() -> Self {
        Request {
            target: None,
            inherit: false,
            timeout: None,
            body: None,
            metadata: None,
            blob: None,
            context: None,
            capabilities: vec![],
        }
    }
    /// Start building a new Request with the Address of the target. In order
    /// to successfully send, you must still fill out at least the `body` field
    /// by calling `body()` or `try_body()` next.
    pub fn to<T>(target: T) -> Self
    where
        T: Into<Address>,
    {
        Request {
            target: Some(target.into()),
            inherit: false,
            timeout: None,
            body: None,
            metadata: None,
            blob: None,
            context: None,
            capabilities: vec![],
        }
    }
    /// Set the target [`Address`] that this request will go to.
```

```
pub fn target<T>(mut self, target: T) -> Self
where
    T: Into<Address>,
{
    self.target = Some(target.into());
    self
}
/// Set whether this request will "inherit" the source / context / blob
/// of the request that this process most recently received. The purpose
/// of inheritance, in this setting, is twofold:
///
/// One, setting inherit to `true` and not attaching a `LazyLoadBlob` will result
/// in the previous request's blob being attached to this request. This
/// is useful for optimizing performance of middleware and other chains of
/// requests that can pass large quantities of data through multiple
/// processes without repeatedly pushing it across the WASM boundary.
///
/// *Note that if the blob of this request is set separately, this flag
/// will not override it.*
///
/// Two, setting inherit to `true` and *not expecting a response* will lead
/// to the previous request's sender receiving the potential response to
/// *this* request. This will only happen if the previous request's sender
/// was expecting a response. This behavior chains, such that many processes
/// could handle inheriting requests while passing the ultimate response back
/// to the very first requester.
pub fn inherit(mut self, inherit: bool) -> Self {
    self.inherit = inherit;
    self
}
/// Set whether this request expects a response, and provide a timeout value
/// (in seconds) within which that response must be received. The sender will
/// receive an error message with this request stored within it if the
/// timeout is triggered.
pub fn expects_response(mut self, timeout: u64) -> Self {
    self.timeout = Some(timeout);
    self
}
/// Set the IPC body (Inter-Process Communication) value for this message. This field
/// is mandatory. An IPC body is simply a vector of bytes. Process developers are
/// responsible for architecting the serialization/derserialization strategy
/// for these bytes, but the simplest and most common strategy is just to use
/// a JSON spec that gets stored in bytes as a UTF-8 string.
///
/// If the serialization strategy is complex, it's best to define it as an impl
/// of [`TryInto`] on your IPC body type, then use `try_body()` instead of this.
pub fn body<T>(mut self, body: T) -> Self
where
    T: Into<Vec<u8>>,
{
    self.body = Some(body.into());
    self
}
/// Set the IPC body (Inter-Process Communication) value for this message, using a
/// type that's got an implementation of [`TryInto`] for `Vec<u8>`. It's best
/// to define an IPC body type within your app, then implement TryFrom/TryInto for
/// all IPC body serialization/deserialization.
pub fn try_body<T>(mut self, body: T) -> anyhow::Result<Self>
where
    T: TryInto<Vec<u8>, Error = anyhow::Error>,
{
    self.body = Some(body.try_into()?);
    Ok(self)
}
/// Set the metdata field for this request. Metadata is simply a [`String`].
/// Metadata should usually be used for middleware and other message-passing
```

```
/// situations that require the original IPC body and blob to be preserved.
/// As such, metadata should not always be expected to reach the final destination
/// of this request unless the full chain of behavior is known / controlled by
/// the developer.
pub fn metadata(mut self, metadata: &str) -> Self {
    self.metadata = Some(metadata.to_string());
    self
}
/// Set the blob of this request. A [`LazyLoadBlob`] holds bytes and an optional
/// MIME type.
///
/// The purpose of having a blob field distinct from the IPC body field is to enable
/// performance optimizations in all sorts of situations. LazyLoadBlobs are only brought
/// across the runtime<>WASM boundary if the process calls `get_blob()`, and this
/// saves lots of work in data-intensive pipelines.
///
/// LazyLoadBlobs also provide a place for less-structured data, such that an IPC body type
/// can be quickly locked in and upgraded within an app-protocol without breaking
/// changes, while still allowing freedom to adjust the contents and shape of a
/// blob. IPC body formats should be rigorously defined.
pub fn blob(mut self, blob: LazyLoadBlob) -> Self {
    self.blob = Some(blob);
    self
}
/// Set the blob's MIME type. If a blob has not been set, it will be set here
/// as an empty vector of bytes. If it has been set, the MIME type will be replaced
/// or created.
pub fn blob_mime(mut self, mime: &str) -> Self {
    if self.blob.is_none() {
        self.blob = Some(LazyLoadBlob {
            mime: Some(mime.to_string()),
            bytes: vec![],
        });
        self
    } else {
        self.blob = Some(LazyLoadBlob {
            mime: Some(mime.to_string()),
            bytes: self.blob.unwrap().bytes,
        });
        self
    }
}
/// Set the blob's bytes. If a blob has not been set, it will be set here with
/// no MIME type. If it has been set, the bytes will be replaced with these bytes.
pub fn blob_bytes<T>(mut self, bytes: T) -> Self
where
    T: Into<Vec<u8>>,
{
    if self.blob.is_none() {
        self.blob = Some(LazyLoadBlob {
            mime: None,
            bytes: bytes.into(),
        });
        self
    } else {
        self.blob = Some(LazyLoadBlob {
            mime: self.blob.unwrap().mime,
            bytes: bytes.into(),
        });
        self
    }
}
/// Set the blob's bytes with a type that implements `TryInto<Vec<u8>>`
/// and may or may not successfully be set.
pub fn try_blob_bytes<T>(mut self, bytes: T) -> anyhow::Result<Self>
where
```

```
    T: TryInto<Vec<u8>, Error = anyhow::Error>,
{
    if self.blob.is_none() {
        self.blob = Some(LazyLoadBlob {
            mime: None,
            bytes: bytes.try_into()?,
        });
        Ok(self)
    } else {
        self.blob = Some(LazyLoadBlob {
            mime: self.blob.unwrap().mime,
            bytes: bytes.try_into()?,
        });
        Ok(self)
    }
}
/// Set the context field of the request. A request's context is just another byte
/// vector. The developer should create a strategy for serializing and deserializing
/// contexts.
///
/// Contexts are useful when avoiding "callback hell". When a request is sent, any
/// response or error (timeout, offline node) will be returned with this context.
/// This allows you to chain multiple asynchronous requests with their responses
/// without using complex logic to store information about outstanding requests.
pub fn context<T>(mut self, context: T) -> Self
where
    T: Into<Vec<u8>>,
{
    self.context = Some(context.into());
    self
}
/// Attempt to set the context field of the request with a type that implements
/// `TryInto<Vec<u8>>`. It's best to define a context type within your app,
/// then implement TryFrom/TryInto for all context serialization/deserialization.
pub fn try_context<T>(mut self, context: T) -> anyhow::Result<Self>
where
    T: TryInto<Vec<u8>, Error = anyhow::Error>,
{
    self.context = Some(context.try_into()?);
    Ok(self)
}
/// Attach capabilities to the next request
pub fn capabilities(mut self, capabilities: Vec<Capability>) -> Self {
    self.capabilities = capabilities;
    self
}
/// Attach the capability to message this process to the next message.
pub fn attach_messaging(mut self, our: &Address) {
    self.capabilities.extend(vec![Capability {
        issuer: our.clone(),
        params: "\"messaging\"".to_string(),
    }]);
}
/// Attempt to send the request. This will only fail if the `target` or `body`
/// fields have not been set.
pub fn send(self) -> anyhow::Result<()> {
    if let (Some(target), Some(body)) = (self.target, self.body) {
        crate::send_request(
            &target,
            &crate::kinode::process::standard::Request {
                inherit: self.inherit,
                expects_response: self.timeout,
                body,
                metadata: self.metadata,
                capabilities: self.capabilities,
            },
```

```
                self.context.as_ref(),
                self.blob.as_ref(),
            );
            Ok(())
        } else {
            Err(anyhow::anyhow!("missing fields"))
        }
    }
    /// Attempt to send the request, then await its response or error (timeout, offline node).
    /// This will only fail if the `target` or `body` fields have not been set.
    pub fn send_and_await_response(
        self,
        timeout: u64,
    ) -> anyhow::Result<Result<Message, SendError>> {
        if let (Some(target), Some(body)) = (self.target, self.body) {
            match crate::send_and_await_response(
                &target,
                &crate::kinode::process::standard::Request {
                    inherit: self.inherit,
                    expects_response: Some(timeout),
                    body,
                    metadata: self.metadata,
                    capabilities: self.capabilities,
                },
                self.blob.as_ref(),
            ) {
                Ok((source, message)) => Ok(Ok(wit_message_to_message(source, message))),
                Err(send_err) => Ok(Err(SendError {
                    kind: match send_err.kind {
                        crate::kinode::process::standard::SendErrorKind::Offline => {
                            SendErrorKind::Offline
                        }
                        crate::kinode::process::standard::SendErrorKind::Timeout => {
                            SendErrorKind::Timeout
                        }
                    },
                    message: wit_message_to_message(
                        Address::new("our", ProcessId::new(Some("net"), "distro", "sys")),
                        send_err.message,
                    ),
                    lazy_load_blob: send_err.lazy_load_blob,
                    context: None,
                })),
            }
        } else {
            Err(anyhow::anyhow!("missing fields"))
        }
    }
}

impl Default for Request {
    fn default() -> Self {
        Request::new()
    }
}

src/process_id.rs
=================

pub use crate::ProcessId;
use serde::{Deserialize, Serialize};
use std::hash::{Hash, Hasher};

/// `ProcessId` is defined in the wit bindings, but constructors and methods
/// are defined here. A `ProcessId` contains a process name, a package name,
/// and a publisher node ID.
```

```
impl ProcessId {
    /// Create a new `ProcessId`. If `process_name` is left as None, this will generate
    /// a random u64 number, convert to string, and store that as the name.
    pub fn new(process_name: Option<&str>, package_name: &str, publisher_node: &str) -> Self {
        ProcessId {
            process_name: process_name
                .unwrap_or(&rand::random::<u64>().to_string())
                .into(),
            package_name: package_name.into(),
            publisher_node: publisher_node.into(),
        }
    }
    /// Read the process name from a `ProcessId`.
    pub fn process(&self) -> &str {
        &self.process_name
    }
    /// Read the package name from a `ProcessId`.
    pub fn package(&self) -> &str {
        &self.package_name
    }
    /// Read the publisher node ID from a `ProcessId`. Note that `ProcessId`
    /// segments are not parsed for validity, and a node ID stored here is
    /// not guaranteed to be a valid ID in the name system, or be connected
    /// to an identity at all.
    pub fn publisher(&self) -> &str {
        &self.publisher_node
    }
}

impl std::str::FromStr for ProcessId {
    type Err = ProcessIdParseError;
    /// Attempts to parse a `ProcessId` from a string. To succeed, the string must contain
    /// exactly 3 segments, separated by colons `:`. The segments must not contain colons.
    /// Please note that while any string without colons will parse successfully
    /// to create a `ProcessId`, not all strings without colons are actually
    /// valid usernames, which the `publisher_node` field of a `ProcessId` will
    /// always in practice be.
    fn from_str(input: &str) -> Result<Self, ProcessIdParseError> {
        // split string on colons into 3 segments
        let mut segments = input.split(':');
        let process_name = segments
            .next()
            .ok_or(ProcessIdParseError::MissingField)?
            .to_string();
        let package_name = segments
            .next()
            .ok_or(ProcessIdParseError::MissingField)?
            .to_string();
        let publisher_node = segments
            .next()
            .ok_or(ProcessIdParseError::MissingField)?
            .to_string();
        if segments.next().is_some() {
            return Err(ProcessIdParseError::TooManyColons);
        }
        Ok(ProcessId {
            process_name,
            package_name,
            publisher_node,
        })
    }
}

impl Serialize for ProcessId {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
```

```
        S: serde::ser::Serializer,
    {
        format!("{}", self).serialize(serializer)
    }
}

impl<'a> Deserialize<'a> for ProcessId {
    fn deserialize<D>(deserializer: D) -> Result<ProcessId, D::Error>
    where
        D: serde::de::Deserializer<'a>,
    {
        let s = String::deserialize(deserializer)?;
        s.parse().map_err(serde::de::Error::custom)
    }
}

impl Hash for ProcessId {
    fn hash<H: Hasher>(&self, state: &mut H) {
        self.process_name.hash(state);
        self.package_name.hash(state);
        self.publisher_node.hash(state);
    }
}

impl Eq for ProcessId {}

impl From<(&str, &str, &str)> for ProcessId {
    fn from(input: (&str, &str, &str)) -> Self {
        ProcessId::new(Some(input.0), input.1, input.2)
    }
}

impl std::fmt::Display for ProcessId {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{}:{}:{}",
            self.process_name, self.package_name, self.publisher_node
        )
    }
}

impl PartialEq for ProcessId {
    fn eq(&self, other: &Self) -> bool {
        self.process_name == other.process_name
            && self.package_name == other.package_name
            && self.publisher_node == other.publisher_node
    }
}

impl PartialEq<&str> for ProcessId {
    fn eq(&self, other: &&str) -> bool {
        &self.to_string() == other
    }
}

impl PartialEq<ProcessId> for &str {
    fn eq(&self, other: &ProcessId) -> bool {
        self == &other.to_string()
    }
}

#[derive(Debug)]
pub enum ProcessIdParseError {
    TooManyColons,
    MissingField,
```

```
}

impl std::fmt::Display for ProcessIdParseError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{}",
            match self {
                ProcessIdParseError::TooManyColons => "Too many colons in ProcessId string",
                ProcessIdParseError::MissingField => "Missing field in ProcessId string",
            }
        )
    }
}

impl std::error::Error for ProcessIdParseError {
    fn description(&self) -> &str {
        match self {
            ProcessIdParseError::TooManyColons => "Too many colons in ProcessId string",
            ProcessIdParseError::MissingField => "Missing field in ProcessId string",
        }
    }
}

src/net.rs
==========

use crate::*;

//
// Networking protocol types
//

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct Identity {
    pub name: NodeId,
    pub networking_key: String,
    pub ws_routing: Option<(String, u16)>,
    pub allowed_routers: Vec<NodeId>,
}

/// Must be parsed from message pack vector.
/// all Get actions must be sent from local process. used for debugging
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum NetAction {
    /// Received from a router of ours when they have a new pending passthrough for us.
    /// We should respond (if we desire) by using them to initialize a routed connection
    /// with the NodeId given.
    ConnectionRequest(NodeId),
    /// can only receive from trusted source, for now just ourselves locally,
    /// in the future could get from remote provider
    KnsUpdate(KnsUpdate),
    KnsBatchUpdate(Vec<KnsUpdate>),
    /// get a list of peers we are connected to
    GetPeers,
    /// get the [`Identity`] struct for a single peer
    GetPeer(String),
    /// get the [`NodeId`] associated with a given namehash, if any
    GetName(String),
    /// get a user-readable diagnostics string containing networking inforamtion
    GetDiagnostics,
    /// sign the attached blob payload, sign with our node's networking key.
    /// **only accepted from our own node**
    /// **the source [`Address`] will always be prepended to the payload**
    Sign,
    /// given a message in blob payload, verify the message is signed by
```

```
    /// the given source. if the signer is not in our representation of
    /// the PKI, will not verify.
    /// **the `from` [`Address`] will always be prepended to the payload**
    Verify {
        from: Address,
        signature: Vec<u8>,
    },
}

/// For now, only sent in response to a ConnectionRequest.
/// Must be parsed from message pack vector
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum NetResponse {
    Accepted(NodeId),
    Rejected(NodeId),
    /// response to [`NetAction::GetPeers`]
    Peers(Vec<Identity>),
    /// response to [`NetAction::GetPeer`]
    Peer(Option<Identity>),
    /// response to [`NetAction::GetName`]
    Name(Option<String>),
    /// response to [`NetAction::GetDiagnostics`]. a user-readable string.
    Diagnostics(String),
    /// response to [`NetAction::Sign`]. contains the signature in blob
    Signed,
    /// response to [`NetAction::Verify`]. boolean indicates whether
    /// the signature was valid or not. note that if the signer node
    /// cannot be found in our representation of PKI, this will return false,
    /// because we cannot find the networking public key to verify with.
    Verified(bool),
}

#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct KnsUpdate {
    pub name: String, // actual username / domain name
    pub owner: String,
    pub node: String, // hex namehash of node
    pub public_key: String,
    pub ip: String,
    pub port: u16,
    pub routers: Vec<String>,
}

//
// Helpers
//

pub fn sign<T>(message: T) -> Result<Vec<u8>, SendError>
where
    T: Into<Vec<u8>>,
{
    Request::to(("our", "net", "distro", "sys"))
        .body(rmp_serde::to_vec(&NetAction::Sign).unwrap())
        .blob_bytes(message.into())
        .send_and_await_response(30)
        .unwrap()
        .map(|_resp| get_blob().unwrap().bytes)
}

pub fn verify<T, U, V>(from: T, message: U, signature: V) -> Result<bool, SendError>
where
    T: Into<Address>,
    U: Into<Vec<u8>>,
    V: Into<Vec<u8>>,
{
    Request::to(("our", "net", "distro", "sys"))
```

```
        .body(
            rmp_serde::to_vec(&NetAction::Verify {
                from: from.into(),
                signature: signature.into(),
            })
            .unwrap(),
        )
        .blob_bytes(message.into())
        .send_and_await_response(30)
        .unwrap()
        .map(|resp| {
            let Ok(NetResponse::Verified(valid)) =
                rmp_serde::from_slice::<NetResponse>(resp.body())
            else {
                return false;
            };
            valid
        })
}

/// take a DNSwire-formatted node ID from chain and convert it to a String
pub fn dnswire_decode(wire_format_bytes: &[u8]) -> Result<String, DnsDecodeError> {
    let mut i = 0;
    let mut result = Vec::new();

    while i < wire_format_bytes.len() {
        let len = wire_format_bytes[i] as usize;
        if len == 0 {
            break;
        }
        let end = i + len + 1;
        let mut span = match wire_format_bytes.get(i + 1..end) {
            Some(span) => span.to_vec(),
            None => return Err(DnsDecodeError::FormatError),
        };
        span.push('.' as u8);
        result.push(span);
        i = end;
    }

    let flat: Vec<_> = result.into_iter().flatten().collect();

    let name = String::from_utf8(flat).map_err(|e| DnsDecodeError::Utf8Error(e))?;

    // Remove the trailing '.' if it exists (it should always exist)
    if name.ends_with('.') {
        Ok(name[0..name.len() - 1].to_string())
    } else {
        Ok(name)
    }
}

#[derive(Clone, Debug, thiserror::Error)]
pub enum DnsDecodeError {
    Utf8Error(std::string::FromUtf8Error),
    FormatError,
}

impl std::fmt::Display for DnsDecodeError {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match self {
            DnsDecodeError::Utf8Error(e) => write!(f, "UTF-8 error: {}", e),
            DnsDecodeError::FormatError => write!(f, "Format error"),
        }
    }
}
```

src/on_exit.rs
==========

```
use crate::*;

#[derive(Clone, Debug)]
pub enum OnExit {
    None,
    Restart,
    Requests(Vec<Request>),
}

impl OnExit {
    /// Call the kernel to get the current set OnExit behavior
    pub fn get() -> Self {
        match crate::kinode::process::standard::get_on_exit() {
            crate::kinode::process::standard::OnExit::None => OnExit::None,
            crate::kinode::process::standard::OnExit::Restart => OnExit::Restart,
            crate::kinode::process::standard::OnExit::Requests(reqs) => {
                let mut requests: Vec<Request> = Vec::with_capacity(reqs.len());
                for req in reqs {
                    requests.push(Request {
                        target: Some(req.0),
                        inherit: req.1.inherit,
                        timeout: req.1.expects_response,
                        body: Some(req.1.body),
                        metadata: req.1.metadata,
                        blob: req.2,
                        context: None,
                        capabilities: req.1.capabilities, // TODO double check
                    });
                }
                OnExit::Requests(requests)
            }
        }
    }
    /// Check if this OnExit is None
    pub fn is_none(&self) -> bool {
        match self {
            OnExit::None => true,
            OnExit::Restart => false,
            OnExit::Requests(_) => false,
        }
    }
    /// Check if this OnExit is Restart
    pub fn is_restart(&self) -> bool {
        match self {
            OnExit::None => false,
            OnExit::Restart => true,
            OnExit::Requests(_) => false,
        }
    }
    /// Check if this OnExit is Requests
    pub fn is_requests(&self) -> bool {
        match self {
            OnExit::None => false,
            OnExit::Restart => false,
            OnExit::Requests(_) => true,
        }
    }
    /// Get the Requests variant of this OnExit, if it is one
    pub fn get_requests(&self) -> Option<&[Request]> {
        match self {
            OnExit::None => None,
            OnExit::Restart => None,
            OnExit::Requests(reqs) => Some(reqs),
```

```
        }
    }
    /// Add a request to this OnExit if it is Requests, fail otherwise
    pub fn add_request(self, new: Request) -> anyhow::Result<()> {
        match self {
            OnExit::None => Err(anyhow::anyhow!("cannot add request to None")),
            OnExit::Restart => Err(anyhow::anyhow!("cannot add request to Restart")),
            OnExit::Requests(mut reqs) => {
                reqs.push(new);
                Ok(())
            }
        }
    }
    /// Set the OnExit behavior for this process
    pub fn set(self) -> anyhow::Result<()> {
        crate::kinode::process::standard::set_on_exit(&self._to_standard()?);
        Ok(())
    }
    /// Convert this OnExit to the kernel's OnExit type
    pub fn _to_standard(self) -> anyhow::Result<crate::kinode::process::standard::OnExit> {
        match self {
            OnExit::None => Ok(crate::kinode::process::standard::OnExit::None),
            OnExit::Restart => Ok(crate::kinode::process::standard::OnExit::Restart),
            OnExit::Requests(reqs) => {
                let mut kernel_reqs: Vec<(
                    Address,
                    kinode::process::standard::Request,
                    Option<LazyLoadBlob>,
                )> = Vec::with_capacity(reqs.len());
                for req in reqs {
                    kernel_reqs.push((
                        req.target
                            .ok_or(anyhow::anyhow!("request without target given"))?,
                        kinode::process::standard::Request {
                            inherit: req.inherit,
                            expects_response: None,
                            body: req
                                .body
                                .ok_or(anyhow::anyhow!("request without body given"))?,
                            metadata: req.metadata,
                            capabilities: req.capabilities, // TODO double check
                        },
                        req.blob,
                    ));
                }
                Ok(crate::kinode::process::standard::OnExit::Requests(
                    kernel_reqs,
                ))
            }
        }
    }
}

src/http.rs
===========

use crate::kernel_types::MessageType;
use crate::vfs::{FileType, VfsAction, VfsRequest, VfsResponse};
use crate::{
    get_blob, Address, LazyLoadBlob as KiBlob, Message, Request as KiRequest,
    Response as KiResponse,
};
pub use http::*;
use serde::{Deserialize, Serialize};
use std::collections::{HashMap, VecDeque};
use std::path::Path;
```

```
use std::str::FromStr;
use thiserror::Error;

//
// these types are a copy of the types used in http module of runtime.
//

/// HTTP Request type that can be shared over WASM boundary to apps.
/// This is the one you receive from the `http_server:distro:sys` service.
#[derive(Debug, Serialize, Deserialize)]
pub enum HttpServerRequest {
    Http(IncomingHttpRequest),
    /// Processes will receive this kind of request when a client connects to them.
    /// If a process does not want this websocket open, they should issue a *request*
    /// containing a [`type@HttpServerAction::WebSocketClose`] message and this channel ID.
    WebSocketOpen {
        path: String,
        channel_id: u32,
    },
    /// Processes can both SEND and RECEIVE this kind of request
    /// (send as [`type@HttpServerAction::WebSocketPush`]).
    /// When received, will contain the message bytes as lazy_load_blob.
    WebSocketPush {
        channel_id: u32,
        message_type: WsMessageType,
    },
    /// Receiving will indicate that the client closed the socket. Can be sent to close
    /// from the server-side, as [`type@HttpServerAction::WebSocketClose`].
    WebSocketClose(u32),
}

#[derive(Debug, Serialize, Deserialize)]
pub struct IncomingHttpRequest {
    source_socket_addr: Option<String>, // will parse to SocketAddr
    method: String,                     // will parse to http::Method
    url: String,                        // will parse to url::Url
    bound_path: String,                 // the matching path that was bound
    headers: HashMap<String, String>,   // will parse to http::HeaderMap
    url_params: HashMap<String, String>,
    query_params: HashMap<String, String>,
    // BODY is stored in the lazy_load_blob, as bytes
}

/// HTTP Response type that can be shared over WASM boundary to apps.
/// Respond to [`IncomingHttpRequest`] with this type.
#[derive(Debug, Serialize, Deserialize)]
pub struct HttpResponse {
    pub status: u16,
    pub headers: HashMap<String, String>,
    // BODY is stored in the lazy_load_blob, as bytes
}

/// Request type sent to `http_server:distro:sys` in order to configure it.
/// You can also send [`type@HttpServerAction::WebSocketPush`], which
/// allows you to push messages across an existing open WebSocket connection.
///
/// If a response is expected, all HttpServerActions will return a Response
/// with the shape Result<(), HttpServerActionError> serialized to JSON.
#[derive(Debug, Serialize, Deserialize)]
pub enum HttpServerAction {
    /// Bind expects a lazy_load_blob if and only if `cache` is TRUE. The lazy_load_blob should
    /// be the static file to serve at this path.
    Bind {
        path: String,
        /// Set whether the HTTP request needs a valid login cookie, AKA, whether
        /// the user needs to be logged in to access this path.
```

```
    authenticated: bool,
    /// Set whether requests can be fielded from anywhere, or only the loopback address.
    local_only: bool,
    /// Set whether to bind the lazy_load_blob statically to this path. That is, take the
    /// lazy_load_blob bytes and serve them as the response to any request to this path.
    cache: bool,
},
/// SecureBind expects a lazy_load_blob if and only if `cache` is TRUE. The lazy_load_blob should
/// be the static file to serve at this path.
///
/// SecureBind is the same as Bind, except that it forces requests to be made from
/// the unique subdomain of the process that bound the path. These requests are
/// *always* authenticated, and *never* local_only. The purpose of SecureBind is to
/// serve elements of an app frontend or API in an exclusive manner, such that other
/// apps installed on this node cannot access them. Since the subdomain is unique, it
/// will require the user to be logged in separately to the general domain authentication.
SecureBind {
    path: String,
    /// Set whether to bind the lazy_load_blob statically to this path. That is, take the
    /// lazy_load_blob bytes and serve them as the response to any request to this path.
    cache: bool,
},
/// Unbind a previously-bound HTTP path
Unbind { path: String },
/// Bind a path to receive incoming WebSocket connections.
/// Doesn't need a cache since does not serve assets.
WebSocketBind {
    path: String,
    authenticated: bool,
    encrypted: bool,
    extension: bool,
},
/// SecureBind is the same as Bind, except that it forces new connections to be made
/// from the unique subdomain of the process that bound the path. These are *always*
/// authenticated. Since the subdomain is unique, it will require the user to be
/// logged in separately to the general domain authentication.
WebSocketSecureBind {
    path: String,
    encrypted: bool,
    extension: bool,
},
/// Unbind a previously-bound WebSocket path
WebSocketUnbind { path: String },
/// Processes will RECEIVE this kind of request when a client connects to them.
/// If a process does not want this websocket open, they should issue a *request*
/// containing a [`type@HttpServerAction::WebSocketClose`] message and this channel ID.
WebSocketOpen { path: String, channel_id: u32 },
/// When sent, expects a lazy_load_blob containing the WebSocket message bytes to send.
WebSocketPush {
    channel_id: u32,
    message_type: WsMessageType,
},
/// When sent, expects a `lazy_load_blob` containing the WebSocket message bytes to send.
/// Modifies the `lazy_load_blob` by placing into `WebSocketExtPushData` with id taken from
/// this `KernelMessage` and `kinode_message_type` set to `desired_reply_type`.
WebSocketExtPushOutgoing {
    channel_id: u32,
    message_type: WsMessageType,
    desired_reply_type: MessageType,
},
/// For communicating with the ext.
/// Kinode's http_server sends this to the ext after receiving `WebSocketExtPushOutgoing`.
/// Upon receiving reply with this type from ext, http_server parses, setting:
/// * id as given,
/// * message type as given (Request or Response),
/// * body as HttpServerRequest::WebSocketPush,
```

```
    /// * blob as given.
    WebSocketExtPushData {
        id: u64,
        kinode_message_type: MessageType,
        blob: Vec<u8>,
    },
    /// Sending will close a socket the process controls.
    WebSocketClose(u32),
}

/// The possible message types for WebSocketPush. Ping and Pong are limited to 125 bytes
/// by the WebSockets protocol. Text will be sent as a Text frame, with the lazy_load_blob bytes
/// being the UTF-8 encoding of the string. Binary will be sent as a Binary frame containing
/// the unmodified lazy_load_blob bytes.
#[derive(Debug, PartialEq, Serialize, Deserialize)]
pub enum WsMessageType {
    Text,
    Binary,
    Ping,
    Pong,
    Close,
}

/// Part of the Response type issued by http_server
#[derive(Error, Debug, Serialize, Deserialize)]
pub enum HttpServerError {
    #[error(
        "http_server: request could not be parsed to HttpServerAction: {}.",
        req
    )]
    BadRequest { req: String },
    #[error("http_server: action expected lazy_load_blob")]
    NoBlob,
    #[error("http_server: path binding error: {:?}", error)]
    PathBindError { error: String },
    #[error("http_server: WebSocket error: {:?}", error)]
    WebSocketPushError { error: String },
}

/// Structure sent from client websocket to this server upon opening a new connection.
/// After this is sent, depending on the `encrypted` flag, the channel will either be
/// open to send and receive plaintext messages or messages encrypted with a symmetric
/// key derived from the JWT.
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct WsRegister {
    pub auth_token: String,
    pub target_process: String,
    pub encrypted: bool, // TODO symmetric key exchange here if true
}

/// Structure sent from this server to client websocket upon opening a new connection.
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct WsRegisterResponse {
    pub channel_id: u32,
    // TODO symmetric key exchange here
}

#[derive(Debug, Serialize, Deserialize)]
pub struct JwtClaims {
    pub username: String,
    pub expiration: u64,
}

impl HttpServerRequest {
    /// Parse a byte slice into an HttpServerRequest.
    pub fn from_bytes(bytes: &[u8]) -> serde_json::Result<Self> {
```

```
        serde_json::from_slice(bytes)
    }

    /// Filter the general-purpose [`HttpServerRequest`], which contains HTTP requests
    /// and WebSocket messages, into just the HTTP request. Consumes the original request
    /// and returns `None` if the request was WebSocket-related.
    pub fn request(self) -> Option<IncomingHttpRequest> {
        match self {
            HttpServerRequest::Http(req) => Some(req),
            _ => None,
        }
    }
}

impl IncomingHttpRequest {
    pub fn url(&self) -> anyhow::Result<url::Url> {
        url::Url::parse(&self.url).map_err(|e| anyhow::anyhow!("couldn't parse url: {:?}", e))
    }

    pub fn method(&self) -> anyhow::Result<http::Method> {
        http::Method::from_bytes(self.method.as_bytes())
            .map_err(|e| anyhow::anyhow!("couldn't parse method: {:?}", e))
    }

    pub fn source_socket_addr(&self) -> anyhow::Result<std::net::SocketAddr> {
        match &self.source_socket_addr {
            Some(addr) => addr
                .parse()
                .map_err(|_| anyhow::anyhow!("Invalid format for socket address: {}", addr)),
            None => Err(anyhow::anyhow!("No source socket address provided")),
        }
    }

    /// Returns the path that was originally bound, with an optional prefix stripped.
    /// The prefix would normally be the process ID as a &str, but it could be anything.
    pub fn bound_path(&self, process_id_to_strip: Option<&str>) -> &str {
        match process_id_to_strip {
            Some(process_id) => self
                .bound_path
                .strip_prefix(&format!("/{}", process_id))
                .unwrap_or(&self.bound_path),
            None => &self.bound_path,
        }
    }

    pub fn path(&self) -> anyhow::Result<String> {
        let url = url::Url::parse(&self.url)?;
        // skip the first path segment, which is the process ID.
        let path = url
            .path_segments()
            .ok_or(anyhow::anyhow!("url path missing process ID!"))?
            .skip(1)
            .collect::<Vec<&str>>()
            .join("/");
        Ok(format!("/{}", path))
    }

    pub fn headers(&self) -> HeaderMap {
        let mut header_map = HeaderMap::new();
        for (key, value) in self.headers.iter() {
            let key_bytes = key.as_bytes();
            let Ok(key_name) = HeaderName::from_bytes(key_bytes) else {
                continue;
            };
            let Ok(value_header) = HeaderValue::from_str(&value) else {
                continue;
```

```
            };
            header_map.insert(key_name, value_header);
        }
        header_map
    }

    pub fn url_params(&self) -> &HashMap<String, String> {
        &self.url_params
    }

    pub fn query_params(&self) -> &HashMap<String, String> {
        &self.query_params
    }
}

/// Request type that can be shared over WASM boundary to apps.
/// This is the one you send to the `http_client:distro:sys` service.
#[derive(Debug, Serialize, Deserialize)]
pub enum HttpClientAction {
    Http(OutgoingHttpRequest),
    WebSocketOpen {
        url: String,
        headers: HashMap<String, String>,
        channel_id: u32,
    },
    WebSocketPush {
        channel_id: u32,
        message_type: WsMessageType,
    },
    WebSocketClose {
        channel_id: u32,
    },
}

/// HTTP Request type that can be shared over WASM boundary to apps.
/// This is the one you send to the `http_client:distro:sys` service.
#[derive(Debug, Serialize, Deserialize)]
pub struct OutgoingHttpRequest {
    pub method: String,          // must parse to http::Method
    pub version: Option<String>, // must parse to http::Version
    pub url: String,             // must parse to url::Url
    pub headers: HashMap<String, String>,
    // BODY is stored in the lazy_load_blob, as bytes
    // TIMEOUT is stored in the message expect_response
}

/// WebSocket Client Request type that can be shared over WASM boundary to apps.
/// This comes from an open websocket client connection in the `http_client:distro:sys` service.
#[derive(Debug, Serialize, Deserialize)]
pub enum HttpClientRequest {
    WebSocketPush {
        channel_id: u32,
        message_type: WsMessageType,
    },
    WebSocketClose {
        channel_id: u32,
    },
}

/// HTTP Client Response type that can be shared over WASM boundary to apps.
/// This is the one you receive from the `http_client:distro:sys` service.
#[derive(Debug, Serialize, Deserialize)]
pub enum HttpClientResponse {
    Http(HttpResponse),
    WebSocketAck,
}
```

```
#[derive(Error, Debug, Serialize, Deserialize)]
pub enum HttpClientError {
    // HTTP errors, may also be applicable to OutgoingWebSocketClientRequest::Open
    #[error("http_client: request is not valid HttpClientRequest: {}.", req)]
    BadRequest { req: String },
    #[error("http_client: http method not supported: {}", method)]
    BadMethod { method: String },
    #[error("http_client: url could not be parsed: {}", url)]
    BadUrl { url: String },
    #[error("http_client: http version not supported: {}", version)]
    BadVersion { version: String },
    #[error("http_client: failed to execute request {}", error)]
    RequestFailed { error: String },

    // WebSocket errors
    #[error("websocket_client: failed to open connection {}", url)]
    WsOpenFailed { url: String },
    #[error("websocket_client: failed to send message {}", req)]
    WsPushFailed { req: String },
    #[error("websocket_client: failed to close connection {}", channel_id)]
    WsCloseFailed { channel_id: u32 },
}

/// Register a new path with the HTTP server. This will cause the HTTP server to
/// forward any requests on this path to the calling process.
pub fn bind_http_path<T>(
    path: T,
    authenticated: bool,
    local_only: bool,
) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpServerAction::Bind {
                path: path.into(),
                authenticated,
                local_only,
                cache: false,
            })
            .unwrap(),
        )
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

/// Register a new path with the HTTP server, and serve a static file from it.
/// The server will respond to GET requests on this path with the given file.
pub fn bind_http_static_path<T>(
    path: T,
    authenticated: bool,
    local_only: bool,
    content_type: Option<String>,
```

```
    content: Vec<u8>,
) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpServerAction::Bind {
                path: path.into(),
                authenticated,
                local_only,
                cache: true,
            })
            .unwrap(),
        )
        .blob(crate::kinode::process::standard::LazyLoadBlob {
            mime: content_type,
            bytes: content,
        })
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

pub fn unbind_http_path<T>(path: T) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(serde_json::to_vec(&HttpServerAction::Unbind { path: path.into() }).unwrap())
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

/// Register a WebSockets path with the HTTP server. Your app must do this
/// in order to receive incoming WebSocket connections.
pub fn bind_ws_path<T>(
    path: T,
    authenticated: bool,
    encrypted: bool,
) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
```

```
        .body(
            serde_json::to_vec(&HttpServerAction::WebSocketBind {
                path: path.into(),
                authenticated,
                encrypted,
                extension: false,
            })
            .unwrap(),
        )
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

/// Register a WebSockets path with the HTTP server to send and
/// receive system messages from a runtime extension. Only use
/// this if you are writing a runtime extension.
pub fn bind_ext_path<T>(path: T) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpServerAction::WebSocketBind {
                path: path.into(),
                authenticated: false,
                encrypted: false,
                extension: true,
            })
            .unwrap(),
        )
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

pub fn unbind_ws_path<T>(path: T) -> std::result::Result<(), HttpServerError>
where
    T: Into<String>,
{
    let res = KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(serde_json::to_vec(&HttpServerAction::WebSocketUnbind { path: path.into() }).unwrap())
        .send_and_await_response(5)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpServerError::PathBindError {
```

```
            error: "http_server timed out".to_string(),
        });
    };
    let Ok(resp) = serde_json::from_slice::<std::result::Result<(), HttpServerError>>(&body) else {
        return Err(HttpServerError::PathBindError {
            error: "http_server gave unexpected response".to_string(),
        });
    };
    resp
}

/// Send an HTTP response to the incoming HTTP request.
pub fn send_response(status: StatusCode, headers: Option<HashMap<String, String>>, body: Vec<u8>)
    KiResponse::new()
        .body(
            serde_json::to_vec(&HttpResponse {
                status: status.as_u16(),
                headers: headers.unwrap_or_default(),
            })
            .unwrap(),
        )
        .blob_bytes(body)
        .send()
        .unwrap()
}

/// Fire off an HTTP request. If a timeout is given, the response will
/// come in the main event loop, otherwise none will be given.
///
/// Note that the response type is [`type@HttpClientResponse`], which, if
/// it originated from this request, will be of the variant [`type@HttpClientResponse::Http`].
/// It will need to be parsed and the body of the response will be stored in the LazyLoadBlob.
pub fn send_request(
    method: Method,
    url: url::Url,
    headers: Option<HashMap<String, String>>,
    timeout: Option<u64>,
    body: Vec<u8>,
) {
    let req = KiRequest::to(("our", "http_client", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpClientAction::Http(OutgoingHttpRequest {
                method: method.to_string(),
                version: None,
                url: url.to_string(),
                headers: headers.unwrap_or_default(),
            }))
            .unwrap(),
        )
        .blob_bytes(body);
    if let Some(timeout) = timeout {
        req.expects_response(timeout).send().unwrap()
    } else {
        req.send().unwrap()
    }
}

/// Make an HTTP request using http_client and await its response.
///
/// Returns [`Response`] from the `http` crate if successful, with the body type as bytes.
pub fn send_request_await_response(
    method: Method,
    url: url::Url,
    headers: Option<HashMap<String, String>>,
    timeout: u64,
    body: Vec<u8>,
```

```
) -> std::result::Result<Response<Vec<u8>>, HttpClientError> {
    let res = KiRequest::to(("our", "http_client", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpClientAction::Http(OutgoingHttpRequest {
                method: method.to_string(),
                version: None,
                url: url.to_string(),
                headers: headers.unwrap_or_default(),
            }))
            .map_err(|e| HttpClientError::BadRequest {
                req: format!("{e:?}"),
            })?,
        )
        .blob_bytes(body)
        .send_and_await_response(timeout)
        .unwrap();
    let Ok(Message::Response { body, .. }) = res else {
        return Err(HttpClientError::RequestFailed {
            error: "http_client timed out".to_string(),
        });
    };
    let resp = match serde_json::from_slice::<
        std::result::Result<HttpClientResponse, HttpClientError>,
    >(&body)
    {
        Ok(Ok(HttpClientResponse::Http(resp))) => resp,
        Ok(Ok(HttpClientResponse::WebSocketAck)) => {
            return Err(HttpClientError::RequestFailed {
                error: "http_client gave unexpected response".to_string(),
            })
        }
        Ok(Err(e)) => return Err(e),
        Err(e) => {
            return Err(HttpClientError::RequestFailed {
                error: format!("http_client gave invalid response: {e:?}"),
            })
        }
    };
    let mut http_response = http::Response::builder()
        .status(http::StatusCode::from_u16(resp.status).unwrap_or_default());
    let headers = http_response.headers_mut().unwrap();
    for (key, value) in &resp.headers {
        let Ok(key) = http::header::HeaderName::from_str(key) else {
            return Err(HttpClientError::RequestFailed {
                error: format!("http_client gave invalid header key: {key}"),
            });
        };
        let Ok(value) = http::header::HeaderValue::from_str(value) else {
            return Err(HttpClientError::RequestFailed {
                error: format!("http_client gave invalid header value: {value}"),
            });
        };
        headers.insert(key, value);
    }
    Ok(http_response
        .body(get_blob().unwrap_or_default().bytes)
        .unwrap())
}

pub fn get_mime_type(filename: &str) -> String {
    let file_path = Path::new(filename);

    let extension = file_path
        .extension()
        .and_then(|ext| ext.to_str())
        .unwrap_or("octet-stream");
```

```
    mime_guess::from_ext(extension)
        .first_or_octet_stream()
        .to_string()
}

/// Serve index.html
pub fn serve_index_html(
    our: &Address,
    directory: &str,
    authenticated: bool,
    local_only: bool,
    paths: Vec<&str>,
) -> anyhow::Result<()> {
    KiRequest::to(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&VfsRequest {
            path: format!("/{}/pkg/{}/index.html", our.package_id(), directory),
            action: VfsAction::Read,
        })?)
        .send_and_await_response(5)??;

    let Some(blob) = get_blob() else {
        return Err(anyhow::anyhow!("serve_index_html: no index.html blob"));
    };

    let index = String::from_utf8(blob.bytes)?;

    for path in paths {
        bind_http_static_path(
            path,
            authenticated,
            local_only,
            Some("text/html".to_string()),
            index.to_string().as_bytes().to_vec(),
        )?;
    }

    Ok(())
}

/// Serve static files from a given directory by binding all of them
/// in http_server to their filesystem path.
pub fn serve_ui(
    our: &Address,
    directory: &str,
    authenticated: bool,
    local_only: bool,
    paths: Vec<&str>,
) -> anyhow::Result<()> {
    serve_index_html(our, directory, authenticated, local_only, paths)?;

    let initial_path = format!("{}/pkg/{}", our.package_id(), directory);

    let mut queue = VecDeque::new();
    queue.push_back(initial_path.clone());

    while let Some(path) = queue.pop_front() {
        let Ok(directory_response) = KiRequest::to(("our", "vfs", "distro", "sys"))
            .body(serde_json::to_vec(&VfsRequest {
                path,
                action: VfsAction::ReadDir,
            })?)
            .send_and_await_response(5)?
        else {
            return Err(anyhow::anyhow!("serve_ui: no response for path"));
        };
```

```
        let directory_body = serde_json::from_slice::<VfsResponse>(directory_response.body())?;

        // Determine if it's a file or a directory and handle appropriately
        match directory_body {
            VfsResponse::ReadDir(directory_info) => {
                for entry in directory_info {
                    match entry.file_type {
                        // If it's a file, serve it statically
                        FileType::File => {
                            KiRequest::to(("our", "vfs", "distro", "sys"))
                                .body(serde_json::to_vec(&VfsRequest {
                                    path: entry.path.clone(),
                                    action: VfsAction::Read,
                                })?)
                                .send_and_await_response(5)??;

                            let Some(blob) = get_blob() else {
                                return Err(anyhow::anyhow!(
                                    "serve_ui: no blob for {}",
                                    entry.path
                                ));
                            };

                            let content_type = get_mime_type(&entry.path);

                            bind_http_static_path(
                                entry.path.replace(&initial_path, ""),
                                authenticated, // Must be authenticated
                                local_only,    // Is not local-only
                                Some(content_type),
                                blob.bytes,
                            )?;
                        }
                        FileType::Directory => {
                            // Push the directory onto the queue
                            queue.push_back(entry.path);
                        }
                        _ => {}
                    }
                }
            }
            _ => {
                return Err(anyhow::anyhow!(
                    "serve_ui: unexpected response for path: {:?}",
                    directory_body
                ))
            }
        };
    }

    Ok(())
}

pub fn handle_ui_asset_request(our: &Address, directory: &str, path: &str) -> anyhow::Result<()> {
    let parts: Vec<&str> = path.split(&our.process.to_string()).collect();
    let after_process = parts.get(1).unwrap_or(&"");

    let target_path = format!("{}/{}", directory, after_process.trim_start_matches('/'));

    KiRequest::to(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&VfsRequest {
            path: format!("{}/pkg/{}", our.package_id(), target_path),
            action: VfsAction::Read,
        })?)
        .send_and_await_response(5)??;
```

```
    let mut headers = HashMap::new();
    let content_type = get_mime_type(path);
    headers.insert("Content-Type".to_string(), content_type);

    KiResponse::new()
        .body(
            serde_json::json!(HttpResponse {
                status: 200,
                headers,
            })
            .to_string()
            .as_bytes()
            .to_vec(),
        )
        .inherit(true)
        .send()?;

    Ok(())
}

pub fn send_ws_push(channel_id: u32, message_type: WsMessageType, blob: KiBlob) {
    KiRequest::to(("our", "http_server", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpServerRequest::WebSocketPush {
                channel_id,
                message_type,
            })
            .unwrap(),
        )
        .blob(blob)
        .send()
        .unwrap()
}

pub fn open_ws_connection(
    url: String,
    headers: Option<HashMap<String, String>>,
    channel_id: u32,
) -> std::result::Result<(), HttpClientError> {
    let Ok(Ok(Message::Response { body, .. })) =
        KiRequest::to(("our", "http_client", "distro", "sys"))
            .body(
                serde_json::to_vec(&HttpClientAction::WebSocketOpen {
                    url: url.clone(),
                    headers: headers.unwrap_or(HashMap::new()),
                    channel_id,
                })
                .unwrap(),
            )
            .send_and_await_response(5)
    else {
        return Err(HttpClientError::WsOpenFailed { url });
    };
    match serde_json::from_slice(&body) {
        Ok(Ok(HttpClientResponse::WebSocketAck)) => Ok(()),
        Ok(Err(e)) => Err(e),
        _ => Err(HttpClientError::WsOpenFailed { url }),
    }
}

pub fn send_ws_client_push(channel_id: u32, message_type: WsMessageType, blob: KiBlob) {
    KiRequest::to(("our", "http_client", "distro", "sys"))
        .body(
            serde_json::to_vec(&HttpClientAction::WebSocketPush {
                channel_id,
```

```
                message_type,
            })
            .unwrap(),
        )
        .blob(blob)
        .send()
        .unwrap()
}

pub fn close_ws_connection(channel_id: u32) -> std::result::Result<(), HttpClientError> {
    let Ok(Ok(Message::Response { body, .. })) =
        KiRequest::to(("our", "http_client", "distro", "sys"))
            .body(
                serde_json::json!(HttpClientAction::WebSocketClose { channel_id })
                    .to_string()
                    .as_bytes()
                    .to_vec(),
            )
            .send_and_await_response(5)
    else {
        return Err(HttpClientError::WsCloseFailed { channel_id });
    };
    match serde_json::from_slice(&body) {
        Ok(Ok(HttpClientResponse::WebSocketAck)) => Ok(()),
        Ok(Err(e)) => Err(e),
        _ => Err(HttpClientError::WsCloseFailed { channel_id }),
    }
}
```

src/vfs/mod.rs
==========

```
use crate::{Message, Request};
use serde::{Deserialize, Serialize};
use thiserror::Error;

pub mod directory;
pub mod file;

pub use directory::*;
pub use file::*;

/// IPC body format for requests sent to vfs runtime module
#[derive(Debug, Serialize, Deserialize)]
pub struct VfsRequest {
    /// path is always prepended by package_id, the capabilities of the topmost folder are checked
    /// "/your_package:publisher.os/drive_folder/another_folder_or_file"
    pub path: String,
    pub action: VfsAction,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum VfsAction {
    CreateDrive,
    CreateDir,
    CreateDirAll,
    CreateFile,
    OpenFile { create: bool },
    CloseFile,
    Write,
    WriteAll,
    Append,
    SyncAll,
    Read,
    ReadDir,
    ReadToEnd,
```

```
    ReadExact(u64),
    ReadToString,
    Seek { seek_from: SeekFrom },
    RemoveFile,
    RemoveDir,
    RemoveDirAll,
    Rename { new_path: String },
    Metadata,
    AddZip,
    CopyFile { new_path: String },
    Len,
    SetLen(u64),
    Hash,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64),
}

#[derive(Debug, Serialize, Deserialize, PartialEq)]
pub enum FileType {
    File,
    Directory,
    Symlink,
    Other,
}

#[derive(Debug, Serialize, Deserialize)]
pub struct FileMetadata {
    pub file_type: FileType,
    pub len: u64,
}

#[derive(Debug, Serialize, Deserialize, PartialEq)]
pub struct DirEntry {
    pub path: String,
    pub file_type: FileType,
}

#[derive(Debug, Serialize, Deserialize)]
pub enum VfsResponse {
    Ok,
    Err(VfsError),
    Read,
    SeekFrom(u64),
    ReadDir(Vec<DirEntry>),
    ReadToString(String),
    Metadata(FileMetadata),
    Len(u64),
    Hash([u8; 32]),
}

#[derive(Error, Debug, Serialize, Deserialize)]
pub enum VfsError {
    #[error("vfs: No capability for action {action} at path {path}")]
    NoCap { action: String, path: String },
    #[error("vfs: Bytes blob required for {action} at path {path}")]
    BadBytes { action: String, path: String },
    #[error("vfs: bad request error: {error}")]
    BadRequest { error: String },
    #[error("vfs: error parsing path: {path}, error: {error}")]
    ParseError { error: String, path: String },
    #[error("vfs: IO error: {error}, at path {path}")]
```

```
    IOError { error: String, path: String },
    #[error("vfs: kernel capability channel error: {error}")]
    CapChannelFail { error: String },
    #[error("vfs: Bad JSON blob: {error}")]
    BadJson { error: String },
    #[error("vfs: File not found at path {path}")]
    NotFound { path: String },
    #[error("vfs: Creating directory failed at path: {path}: {error}")]
    CreateDirError { path: String, error: String },
}

#[allow(dead_code)]
impl VfsError {
    pub fn kind(&self) -> &str {
        match *self {
            VfsError::NoCap { .. } => "NoCap",
            VfsError::BadBytes { .. } => "BadBytes",
            VfsError::BadRequest { .. } => "BadRequest",
            VfsError::ParseError { .. } => "ParseError",
            VfsError::IOError { .. } => "IOError",
            VfsError::CapChannelFail { .. } => "CapChannelFail",
            VfsError::BadJson { .. } => "NoJson",
            VfsError::NotFound { .. } => "NotFound",
            VfsError::CreateDirError { .. } => "CreateDirError",
        }
    }
}

/// Metadata of a path, returns file type and length.
pub fn metadata(path: &str, timeout: Option<u64>) -> anyhow::Result<FileMetadata> {
    let timeout = timeout.unwrap_or(5);

    let request = VfsRequest {
        path: path.to_string(),
        action: VfsAction::Metadata,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Metadata(metadata) => Ok(metadata),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Removes a path, if it's either a directory or a file.
pub fn remove_path(path: &str, timeout: Option<u64>) -> anyhow::Result<()> {
    let meta = metadata(path, timeout)?;
    match meta.file_type {
        FileType::Directory => remove_dir(path, timeout),
        FileType::File => remove_file(path, timeout),
        _ => Err(anyhow::anyhow!(
            "vfs: path is not a file or directory: {}",
            path
        )),
    }
}
```

src/vfs/directory.rs
==========

```
use super::{DirEntry, VfsAction, VfsRequest, VfsResponse};
use crate::{Message, Request};

/// Vfs helper struct for a directory.
/// Opening or creating a directory will give you a Result<Directory>.
/// You can call it's impl functions to interact with it.
pub struct Directory {
    pub path: String,
    pub timeout: u64,
}

impl Directory {
    /// Iterates through children of directory, returning a vector of DirEntries.
    /// DirEntries contain the path and file type of each child.
    pub fn read(&self) -> anyhow::Result<Vec<DirEntry>> {
        let request = VfsRequest {
            path: self.path.clone(),
            action: VfsAction::ReadDir,
        };
        let message = Request::new()
            .target(("our", "vfs", "distro", "sys"))
            .body(serde_json::to_vec(&request)?)
            .send_and_await_response(self.timeout)?;

        match message {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<VfsResponse>(&body)?;
                match response {
                    VfsResponse::ReadDir(entries) => Ok(entries),
                    VfsResponse::Err(e) => Err(e.into()),
                    _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
                }
            }
            _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
        }
    }
}

/// Opens or creates a directory at path.
/// If trying to create an existing directory, will just give you the path.
pub fn open_dir(path: &str, create: bool, timeout: Option<u64>) -> anyhow::Result<Directory> {
    let timeout = timeout.unwrap_or(5);
    if !create {
        return Ok(Directory {
            path: path.to_string(),
            timeout,
        });
    }
    let request = VfsRequest {
        path: path.to_string(),
        action: VfsAction::CreateDir,
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(Directory {
```

```
                path: path.to_string(),
                timeout,
            }),
            VfsResponse::Err(e) => Err(e.into()),
            _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
        }
    }
    _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
  }
}

/// Removes a dir at path, errors if path not found or path is not a directory.
pub fn remove_dir(path: &str, timeout: Option<u64>) -> anyhow::Result<()> {
  let timeout = timeout.unwrap_or(5);

  let request = VfsRequest {
    path: path.to_string(),
    action: VfsAction::RemoveDir,
  };

  let message = Request::new()
    .target(("our", "vfs", "distro", "sys"))
    .body(serde_json::to_vec(&request)?)
    .send_and_await_response(timeout)?;

  match message {
    Ok(Message::Response { body, .. }) => {
      let response = serde_json::from_slice::<VfsResponse>(&body)?;
      match response {
        VfsResponse::Ok => Ok(()),
        VfsResponse::Err(e) => Err(e.into()),
        _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
      }
    }
    _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
  }
}
```

src/vfs/file.rs
==========

```
use super::{FileMetadata, SeekFrom, VfsAction, VfsRequest, VfsResponse};
use crate::{get_blob, Message, PackageId, Request};

/// Vfs helper struct for a file.
/// Opening or creating a file will give you a Result<File>.
/// You can call it's impl functions to interact with it.
pub struct File {
  pub path: String,
  pub timeout: u64,
}

impl File {
  /// Reads the entire file, from start position.
  /// Returns a vector of bytes.
  pub fn read(&self) -> anyhow::Result<Vec<u8>> {
    let request = VfsRequest {
      path: self.path.clone(),
      action: VfsAction::Read,
    };
    let message = Request::new()
      .target(("our", "vfs", "distro", "sys"))
      .body(serde_json::to_vec(&request)?)
      .send_and_await_response(self.timeout)?;

    match message {
```

```
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Read => {
                    let data = match get_blob() {
                        Some(bytes) => bytes.bytes,
                        None => return Err(anyhow::anyhow!("vfs: no read blob")),
                    };
                    Ok(data)
                }
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Reads the entire file, from start position, into buffer.
/// Returns the amount of bytes read.
pub fn read_into(&self, buffer: &mut [u8]) -> anyhow::Result<usize> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::Read,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Read => {
                    let data = match get_blob() {
                        Some(bytes) => bytes.bytes,
                        None => return Err(anyhow::anyhow!("vfs: no read blob")),
                    };
                    let len = std::cmp::min(data.len(), buffer.len());
                    buffer[..len].copy_from_slice(&data[..len]);
                    Ok(len)
                }
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Read into buffer from current cursor position
/// Returns the amount of bytes read.
pub fn read_at(&self, buffer: &mut [u8]) -> anyhow::Result<usize> {
    let length = buffer.len();
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::ReadExact(length as u64),
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
```

```
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Read => {
                    let data = match get_blob() {
                        Some(bytes) => bytes.bytes,
                        None => return Err(anyhow::anyhow!("vfs: no read blob")),
                    };
                    let len = std::cmp::min(data.len(), buffer.len());
                    buffer[..len].copy_from_slice(&data[..len]);
                    Ok(len)
                }
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Reads until end of file from current cursor position
/// Returns a vector of bytes.
pub fn read_to_end(&self) -> anyhow::Result<Vec<u8>> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::ReadToEnd,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Read => {
                    let data = match get_blob() {
                        Some(bytes) => bytes.bytes,
                        None => return Err(anyhow::anyhow!("vfs: no read blob")),
                    };
                    Ok(data)
                }
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Reads until end of file from current cursor position, converts to String.
/// Throws error if bytes aren't valid utf-8.
/// Returns a vector of bytes.
pub fn read_to_string(&self) -> anyhow::Result<String> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::ReadToString,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
```

```
            match response {
                VfsResponse::ReadToString(s) => Ok(s),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Write entire slice as the new file.
/// Truncates anything that existed at path before.
pub fn write(&self, buffer: &[u8]) -> anyhow::Result<()> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::Write,
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .blob_bytes(buffer)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(()),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Write buffer to file at current position, overwriting any existing data.
pub fn write_all(&mut self, buffer: &[u8]) -> anyhow::Result<()> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::WriteAll,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .blob_bytes(buffer)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(()),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Write buffer to the end position of file.
pub fn append(&mut self, buffer: &[u8]) -> anyhow::Result<()> {
    let request = VfsRequest {
        path: self.path.clone(),
```

```
        action: VfsAction::Append,
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .blob_bytes(buffer)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(()),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Seek file to position.
/// Returns the new position.
pub fn seek(&mut self, pos: SeekFrom) -> anyhow::Result<u64> {
    let request = VfsRequest {
        path: self.path.clone(),
        action: VfsAction::Seek { seek_from: pos },
    };
    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(self.timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::SeekFrom(new_pos) => Ok(new_pos),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Copies a file to path, returns a new File.
pub fn copy(&mut self, path: &str) -> anyhow::Result<File> {
    let request = VfsRequest {
        path: self.path.to_string(),
        action: VfsAction::CopyFile {
            new_path: path.to_string(),
        },
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(5)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(File {
                    path: path.to_string(),
```

```
                timeout: self.timeout,
            }),
            VfsResponse::Err(e) => Err(e.into()),
            _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
        }
    }
    _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
  }
}

/// Set file length, if given size > underlying file, fills it with 0s.
pub fn set_len(&mut self, size: u64) -> anyhow::Result<()> {
  let request = VfsRequest {
    path: self.path.clone(),
    action: VfsAction::SetLen(size),
  };
  let message = Request::new()
    .target(("our", "vfs", "distro", "sys"))
    .body(serde_json::to_vec(&request)?)
    .send_and_await_response(self.timeout)?;

  match message {
    Ok(Message::Response { body, .. }) => {
      let response = serde_json::from_slice::<VfsResponse>(&body)?;
      match response {
        VfsResponse::Ok => Ok(()),
        VfsResponse::Err(e) => Err(e.into()),
        _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
      }
    }
    _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
  }
}

/// Metadata of a path, returns file type and length.
pub fn metadata(&self) -> anyhow::Result<FileMetadata> {
  let request = VfsRequest {
    path: self.path.clone(),
    action: VfsAction::Metadata,
  };
  let message = Request::new()
    .target(("our", "vfs", "distro", "sys"))
    .body(serde_json::to_vec(&request)?)
    .send_and_await_response(self.timeout)?;

  match message {
    Ok(Message::Response { body, .. }) => {
      let response = serde_json::from_slice::<VfsResponse>(&body)?;
      match response {
        VfsResponse::Metadata(metadata) => Ok(metadata),
        VfsResponse::Err(e) => Err(e.into()),
        _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
      }
    }
    _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
  }
}

/// Syncs path file buffers to disk.
pub fn sync_all(&self) -> anyhow::Result<()> {
  let request = VfsRequest {
    path: self.path.clone(),
    action: VfsAction::SyncAll,
  };
  let message = Request::new()
    .target(("our", "vfs", "distro", "sys"))
```

```
            .body(serde_json::to_vec(&request)?)
            .send_and_await_response(self.timeout)?;

        match message {
            Ok(Message::Response { body, .. }) => {
                let response = serde_json::from_slice::<VfsResponse>(&body)?;
                match response {
                    VfsResponse::Ok => Ok(()),
                    VfsResponse::Err(e) => Err(e.into()),
                    _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
                }
            }
            _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
        }
    }
}

/// Creates a drive with path "/package_id/drive", gives you read and write caps.
/// Will only work on the same package_id as you're calling it from, unless you
/// have root capabilities.
pub fn create_drive(
    package_id: PackageId,
    drive: &str,
    timeout: Option<u64>,
) -> anyhow::Result<String> {
    let timeout = timeout.unwrap_or(5);

    let path = format!("/{}/{}", package_id, drive);
    let res = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&VfsRequest {
            path: path.clone(),
            action: VfsAction::CreateDrive,
        })?)
        .send_and_await_response(timeout)?;

    match res {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(path),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", res)),
    }
}

/// Opens a file at path, if no file at path, creates one if boolean create is true.
pub fn open_file(path: &str, create: bool, timeout: Option<u64>) -> anyhow::Result<File> {
    let timeout = timeout.unwrap_or(5);

    let request = VfsRequest {
        path: path.to_string(),
        action: VfsAction::OpenFile { create },
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
```

```
            match response {
                VfsResponse::Ok => Ok(File {
                    path: path.to_string(),
                    timeout,
                }),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Creates a file at path, if file found at path, truncates it to 0.
pub fn create_file(path: &str, timeout: Option<u64>) -> anyhow::Result<File> {
    let timeout = timeout.unwrap_or(5);
    let request = VfsRequest {
        path: path.to_string(),
        action: VfsAction::CreateFile,
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(File {
                    path: path.to_string(),
                    timeout,
                }),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
    }
}

/// Removes a file at path, errors if path not found or path is not a file.
pub fn remove_file(path: &str, timeout: Option<u64>) -> anyhow::Result<()> {
    let timeout = timeout.unwrap_or(5);

    let request = VfsRequest {
        path: path.to_string(),
        action: VfsAction::RemoveFile,
    };

    let message = Request::new()
        .target(("our", "vfs", "distro", "sys"))
        .body(serde_json::to_vec(&request)?)
        .send_and_await_response(timeout)?;

    match message {
        Ok(Message::Response { body, .. }) => {
            let response = serde_json::from_slice::<VfsResponse>(&body)?;
            match response {
                VfsResponse::Ok => Ok(()),
                VfsResponse::Err(e) => Err(e.into()),
                _ => Err(anyhow::anyhow!("vfs: unexpected response: {:?}", response)),
            }
        }
        _ => Err(anyhow::anyhow!("vfs: unexpected message: {:?}", message)),
```

```
    }
}
```