Haskell eXchange 2022

How to choose Haskell Web Framework

Alyona Antonova

About me

- ~3 years experience in Haskell
  - 2+ years in industrial software development at Serokell
  - ~1 year of self-study + university course
- Web development beginner

Goals

- Promote Haskell as a good web development tool
- Make a life a bit easier for (future) Haskell web developers

Content

Considered web tools – Servant, Yesod, and IHP

Plan:

- Short overview
- Simple app implementation example
- Logging middleware addition
- Final conclusions

Introduce web tools

Servant

briefly

- “A Type-Level Web DSL” (c)
- Typically used for implementing REST APIs
- Represents web API as Haskell type
- Light-weight
- Quite popular

Yesod

briefly

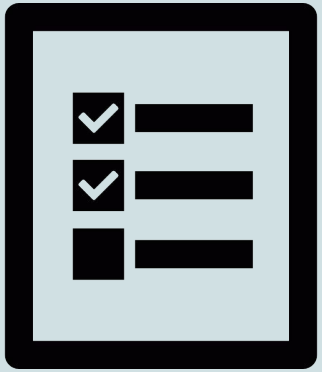
- “Web framework for productive development of type-safe, RESTful, high performance web applications” (c)
- Full-stack
- Modular approach: data store interface (Persistent) and template languages for frontend (Shakespearean Templates) are standalone packages

IHP

briefly

- “IHP is a full-stack framework focused on rapid application development while striving for robust code quality” (c)
- Fully managed environment with Nix
- Automatic code reloading with built-in web server
- GUI for fast prototyping and generating code

Simple app description



TODO-list API

```
data Task = Task { content :: String }
```

TODO-list API

```
data Task = Task { content :: String }
```

`GET /api/task` — get all tasks

`POST /api/task` — create new task

`GET /api/task/{id}` — get task by id

`PUT /api/task/{id}` — update existing task

`DELETE /api/task/{id}` — delete existing task

TODO-list API

```
data Task = Task { content :: String }
```

`GET /api/task` — get all tasks

`POST /api/task` — create new task

`GET /api/task/{id}` — get task by id

`PUT /api/task/{id}` — update existing task

`DELETE /api/task/{id}` — delete existing task

```
> POST /api/task/ '{"content": "wake up"}'
{"content":"wake up","id":1}

> POST /api/task/ '{"content": "go to sleep"}'
{"content":"go to sleep","id":2}

> PUT /api/task/2 '{"content": "do some work!"}'
{"content":"do some work!","id":2}

> GET /api/task/
[{"content":"wake up","id":1},
{"content":"do some work!","id":2}]
```

Database

- **Servant and Yesod:** "*Persistent*" library with Postgres
- **IHP:** Own storage interface with Postgres (the only one available in free version)

How to compare web tools' approaches?

Comparison points

- routes/API syntax
- handlers implementation
- database integration
- extra functionality addition
    - logging middleware
- setting up and running application
- proper documentation
- overall subjective impression

Creating database entity

Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

- SQL table `task`

Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

- SQL table `task`
- `data Task = Task { content :: String }`

Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

- SQL table `task`
- `data Task = Task { content :: String }`
- `type TaskId = Key Task`

Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

- SQL table `task`
- `data Task = Task { content :: String }`
- `type TaskId = Key Task`
- `instance PersistEntity Task where ...`
  - `Entity taskId task :: Entity Task`

    that matches a record in SQL table

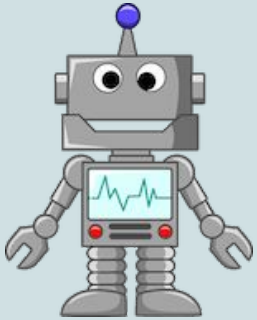
Database entity with Persistent

```
share
  [ mkPersist sqlSettings
  , mkMigrate "migrateAll"
  ]
  [persistLowerCase|
  Task
    content String
    deriving Show
  |]
```

- SQL table `task`
- `data Task = Task { content :: String }`
- `type TaskId = Key Task`
- `instance PersistEntity Task where ...`
  - `Entity taskId task :: Entity Task`

    that matches a record in SQL table
- `migrateAll` function to migrate database

TODO-list by Servant



Servant: API

- API is a Haskell type
- API – sequence of endpoints united by a special combinator `:<|>`
- Endpoint – path segments and combinators

Servant: from API to handlers

API

- `Get, Post, Delete, Put, Patch`
- `Capture “paramName” a`
- `ReqBody contentTypes a`

Servant: from API to handlers

API

- `Get, Post, Delete, Put, Patch`
- `Capture “paramName” a`
- `ReqBody contentTypes a`

Handlers

- Return type
- Argument of type **a**
- Argument of type **a**

Servant: from API to handlers

API

```
"api" :> "task"
    :> ReqBody '[JSON] Task
    :> Post '[JSON] (Entity Task)
```

```
"api" :> "task"
    :> Capture "taskId" TaskId
    :> ReqBody '[JSON] Task
    :> Put '[JSON] (Entity Task)
```

Handlers

```
createEntityTask
    :: Task
    -> Handler (Entity Task)
```

```
replaceEntityTask
    :: TaskId
    -> Task
    -> Handler (Entity Task)
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
        :> Get '[JSON] [Entity Task]
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
       :> Get '[JSON] [Entity Task]
  :<|> "api" :> "task"
       :> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
       :> Get '[JSON] [Entity Task]
  :<|> "api" :> "task"
       :> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)

  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Get '[JSON] (Entity Task)
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
       :> Get '[JSON] [Entity Task]
  :<|> "api" :> "task"
       :> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)

  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Get '[JSON] (Entity Task)
  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> ReqBody '[JSON] Task :> Put '[JSON] (Entity Task)
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
       :> Get '[JSON] [Entity Task]
  :<|> "api" :> "task"
       :> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)

  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Get '[JSON] (Entity Task)
  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> ReqBody '[JSON] Task :> Put '[JSON] (Entity Task)
  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Delete '[JSON] (Entity Task)
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task" :>
       (    Get '[JSON] [Entity Task]
       :<|> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)
       )
  :<|> "api" :> "task" :> Capture "taskId" TaskId :>
       (    Get '[JSON] (Entity Task)
       :<|> ReqBody '[JSON] Task :> Put '[JSON] (Entity Task)
       :<|> Delete '[JSON] (Entity Task)
       )
```

Servant: API definition

```
type TaskListAPI
  =    "api" :> "task"
       :> Get '[JSON] [Entity Task]
  :<|> "api" :> "task"
       :> ReqBody '[JSON] Task :> Post '[JSON] (Entity Task)

  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Get '[JSON] (Entity Task)
  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> ReqBody '[JSON] Task :> Put '[JSON] (Entity Task)
  :<|> "api" :> "task" :> Capture "taskId" TaskId
       :> Delete '[JSON] (Entity Task)
```

Servant: Handlers

```
createEntityTask
  :: Task -> Handler (Entity Task)
createEntityTask task = do
    taskId <- runDB $ insert task
    return $ Entity taskId task
```

```
deleteEntityTaskById :: TaskId -> Handler (Entity Task)
deleteEntityTaskById taskId = do
    entityTask <- getEntityTaskById taskId
    runDB $ delete taskId
    return $ entityTask
```

```
replaceEntityTaskById
  :: TaskId -> Task -> Handler (Entity Task)
replaceEntityTaskById taskId task = do
    _ <- getEntityTaskById taskId
    runDB $ replace taskId task
    return $ Entity taskId task
```

```
getEntityTaskById :: TaskId -> Handler (Entity Task)
getEntityTaskById taskId = do
    maybeTask <- runDB $ get taskId
    case maybeTask of
        Just task -> return $ Entity taskId task
        Nothing   -> throwError err404
```

```
getEntityTasks :: Handler [Entity Task]
getEntityTasks = runDB $ selectList [] []
```

```
-- will be defined later
runDB :: SqlPersistT IO a -> Handler a
```

Servant: Setup & run

```haskell
taskListServer :: Server TaskListAPI
taskListServer
    =    getEntityTasks
    :<|> createEntityTask
    :<|> getEntityTaskById
    :<|> replaceEntityTaskById
    :<|> deleteEntityTaskById


taskListAPI :: Proxy TaskListAPI
taskListAPI = Proxy


taskListApp :: Application
taskListApp = serve taskListAPI taskListServer


main :: IO ()
main = run port taskListApp
    where port = 4000
```

- `Server` – type that handles requests to API
- `Proxy` guides type inference
- `taskListApp` – turns `Server` into WAI `Application`
- `run` creates the working web app

What is the issue?

Servant: Database integration

```haskell
getEntityTasks :: Handler [Entity Task]
getEntityTasks = runDB $ selectList [] []
```

```haskell
runDB :: SqlPersistT IO a -> Handler a
runDB query = do
  -- pseudo-code, don't try to repeat!
  connectToDatabase
  queryResult <- requestDatabase query
  return queryResult
```

Database connection pooling

```
runDB :: SqlPersistT IO a -> Handler a


runSqlPool
  :: SqlPersistT IO a -- query
  -> ConnectionPool   -- pool
  -> ReaderT r IO a
```

Connection pooling – a way to reduce the cost of opening and closing connections by maintaining a “pool” of opened connections that can be reused

Database connection pooling

```
runDB :: SqlPersistT IO a -> Handler a
```

```
runSqlPool
  :: SqlPersistT IO a -- query
  -> ConnectionPool   -- pool
  -> ReaderT r IO a
```

Connection pooling – a way to reduce the cost of opening and closing connections by maintaining a “pool” of opened connections that can be reused

Servant: Using another monad for handlers

```
taskListServer :: Server TaskListAPI
taskListServer
  =    getEntityTasks
       ...

type Server api = ServerT api Handler
```

Servant: Using another monad for handlers

```
taskListServer :: Server TaskListAPI
taskListServer
  =    getEntityTasks
       ...

type Server api = ServerT api Handler
```

```
taskListServerT :: ServerT TaskListAPI App
taskListServerT
  =    getEntityTasks
       ...

data App = ?
```

Servant: Using another monad for handlers

```
newtype App a = App { runApp :: ReaderT Config Handler a }
  deriving
    ( Functor, Applicative, Monad, MonadIO
    , MonadReader Config, MonadError ServerError
    )

data Config
  = Config
  { configPool :: ConnectionPool
  , configPort :: Port
  }
```

Servant: Using another monad for handlers — before

```
runDB :: SqlPersistT IO a -> Handler a
runDB query = do
  -- pseudo-code, don't try to repeat!
  connectToDatabase
  queryResult <- requestDatabase query
  return queryResult
```

```
getEntityTasks
    :: Handler [Entity Task]

getEntityTaskById
    :: TaskId -> Handler (Entity Task)

createEntityTask
    :: Task -> Handler (Entity Task)

replaceEntityTaskById
    :: TaskId -> Task -> Handler (Entity Task)

deleteEntityTaskById
    :: TaskId -> Handler (Entity Task)
```

Servant: Using another monad for handlers — after

```
runDB :: SqlPersistT IO a -> App a
runDB query = do
 pool <- asks configPool
 liftIO $ runSqlPool query pool
```

```
getEntityTasks
    :: App [Entity Task]

getEntityTaskById
    :: TaskId -> App (Entity Task)

createEntityTask
    :: Task -> App (Entity Task)

replaceEntityTaskById
    :: TaskId -> Task -> App (Entity Task)

deleteEntityTaskById
    :: TaskId -> App (Entity Task)
```

Servant: Using another monad for handlers — before

```
taskListServer :: Server TaskListAPI
taskListServer
    =    getEntityTasks
         ...

taskListAPI :: Proxy TaskListAPI
taskListAPI = Proxy


taskListApp :: Application
taskListApp = serve taskListAPI taskListServer
```

```
main :: IO ()
main = run port taskListApp
    where port = 4000
```

Servant: Using another monad for handlers — after

```
taskListServerT :: ServerT TaskListAPI App
taskListServerT
  =    getEntityTasks
       ...

taskListAPI :: Proxy TaskListAPI
taskListAPI = Proxy

appToHandler :: Config -> App a -> Handler a
appToHandler config app = runReaderT (runApp app) config

taskListServer :: Config -> Server TaskListAPI
taskListServer config = hoistServer taskListAPI (appToHandler config) taskListServerT

taskListApp :: Config -> Application
taskListApp config = serve taskListAPI $ taskListServer config
```

Servant: Using another monad for handlers — after

```
main :: IO ()
main = runStderrLoggingT $
  withPostgresqlPool connectionStr connectionsNumber $ \pool ->
    liftIO $ do
      let config = Config
          { configPool = pool
          , configPort = defaultPort
          }

      runSqlPool (runMigration migrateAll) (configPool config)
      run (configPort config) (taskListApp config)
```

TODO-list by Yesod

Yesod: API

```
data TaskList = TaskList Config

mkYesod "TaskList" [parseRoutes|
/api/task          TaskListR GET POST
/api/task/#TaskId TaskR GET DELETE PUT
|]
```

- Route data type
- Parser and render functions
- `YesodDispatch` instance that
  - parse request
  - choose handler
  - run handler
- `Handler` type synonym
- Handlers type definitions

Yesod: Connect route to handler

```
/api/task/#TaskId TaskR PUT


put + TaskR = putTaskR
```

Yesod: Connect route to handler

```
/api/task/#TaskId TaskR PUT

put + TaskR = putTaskR

putTaskR :: TaskId -> Handler Aeson.Value
putTaskR taskId = do
  _ <- getEntityTaskById taskId
  task :: Task <- requireCheckJsonBody
  entityTask <- replaceEntityTask taskId task
  sendStatusJSON ok200 $ toJSON entityTask
```

Yesod: Connect route to handler

```
/api/task/#TaskId TaskR PUT

put + TaskR = putTaskR

putTaskR :: TaskId -> Handler Aeson.Value
putTaskR taskId = do
  _ <- getEntityTaskById taskId
  task :: Task <- requireCheckJsonBody
  entityTask <- replaceEntityTask taskId task
  sendStatusJSON ok200 $ toJSON entityTask
```

```
• Variable not in scope: putTaskR
    :: TaskId -> HandlerFor TaskList res0
  • Perhaps you meant ‘getTaskR’
|
| mkYesod "TaskList" [parseRoutes|
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Yesod: Handlers

Handlers

```
getTaskListR :: Handler Value
getTaskListR = do
 tasks <- getEntityTasks
 sendStatusJSON ok200 $ toJSONList tasks

postTaskListR :: Handler Value
postTaskListR = do
 task :: Task <- requireCheckJsonBody
 entityTask <- createEntityTask task
 sendStatusJSON created201 $ toJSON entityTask

deleteTaskR :: TaskId -> Handler Value
deleteTaskR taskId = do
 entityTask <- getEntityTaskById taskId
 deleteEntityTask taskId
 sendStatusJSON ok200 $ toJSON entityTask
```

DB helpers

```
getEntityTasks :: Handler [Entity Task]
getEntityTasks = runDB $ selectList [] []

getEntityTaskById :: TaskId -> Handler (Entity Task)
getEntityTaskById taskId = do
 task <- runDB $ get404 taskId
 return $ Entity taskId task

createEntityTask :: Task -> Handler (Entity Task)
createEntityTask task = do
 taskId <- runDB $ insert task
 return $ Entity taskId task

deleteEntityTask :: TaskId -> Handler ()
deleteEntityTask = runDB . delete
```

Yesod: Handlers

Handlers

```
getTaskListR :: Handler Value
getTaskListR = do
 tasks <- getEntityTasks
 sendStatusJSON ok200 $ toJSONList tasks

postTaskListR :: Handler Value
postTaskListR = do
 task :: Task <- requireCheckJsonBody
 entityTask <- createEntityTask task
 sendStatusJSON created201 $ toJSON entityTask

deleteTaskR :: TaskId -> Handler Value
deleteTaskR taskId = do
 entityTask <- getEntityTaskById taskId
 deleteEntityTask taskId
 sendStatusJSON ok200 $ toJSON entityTask
```

DB helpers

```
getEntityTasks :: Handler [Entity Task]
getEntityTasks = runDB $ selectList [] []

getEntityTaskById :: TaskId -> Handler (Entity Task)
getEntityTaskById taskId = do
 task <- runDB $ get404 taskId
 return $ Entity taskId task

createEntityTask :: Task -> Handler (Entity Task)
createEntityTask task = do
 taskId <- runDB $ insert task
 return $ Entity taskId task

deleteEntityTask :: TaskId -> Handler ()
deleteEntityTask = runDB . delete
```

Yesod: Database integration

```
data TaskList = TaskList Config

data Config
    = Config
    { configPool :: ConnectionPool
    , configPort :: Port
    }
```

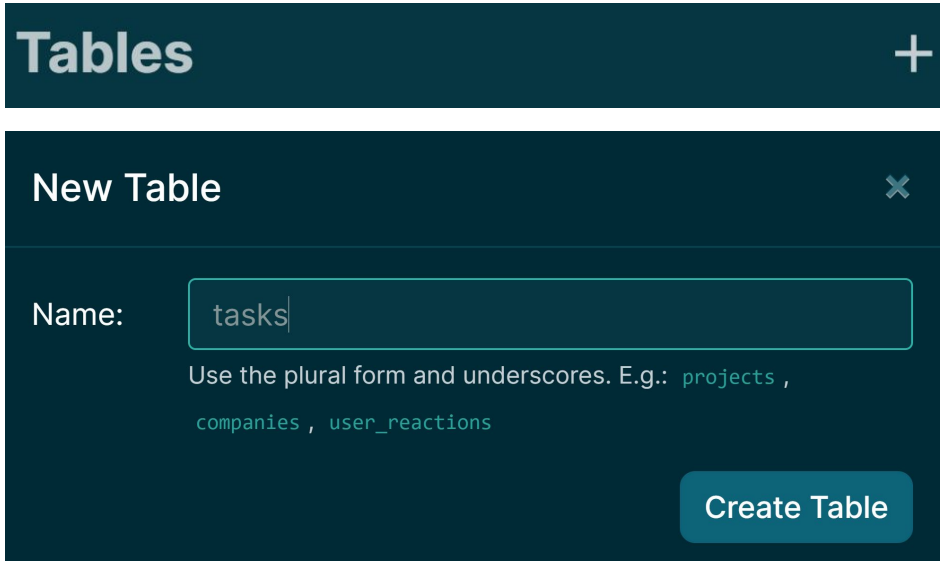
```
instance Yesod TaskList

instance YesodPersist TaskList where
    type YesodPersistBackend TaskList = SqlBackend

    runDB action = do
      TaskList config <- getYesod
      runSqlPool action (configPool config)
```

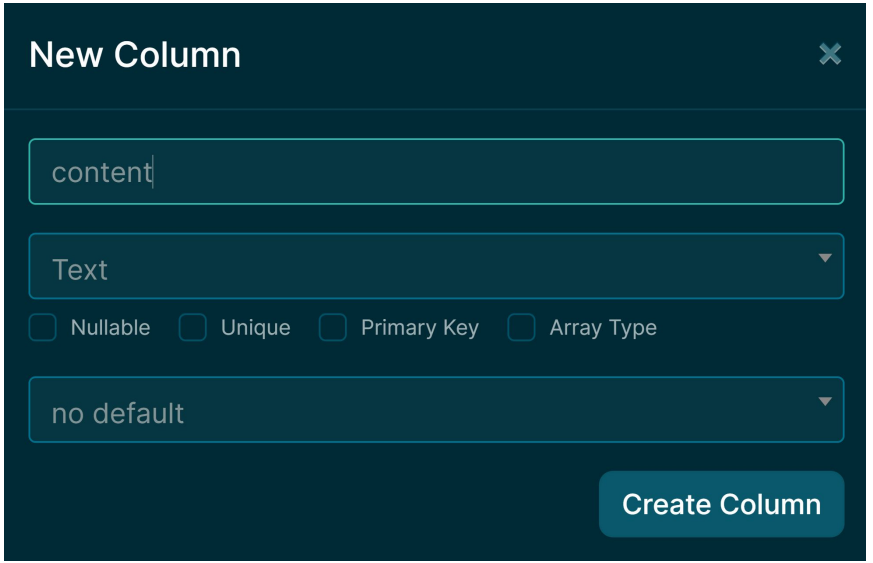
Yesod: Setup & run

```
main :: IO ()
main = runStderrLoggingT $
  withPostgresqlPool connectionStr connectionsNumber $ \pool ->
    liftIO $ do
      let config = Config
        { configPool = pool
        , configPort = defaultPort
        }

      runSqlPool (runMigration migrateAll) (configPool config)
      warp (configPort config) $ TaskList config
```

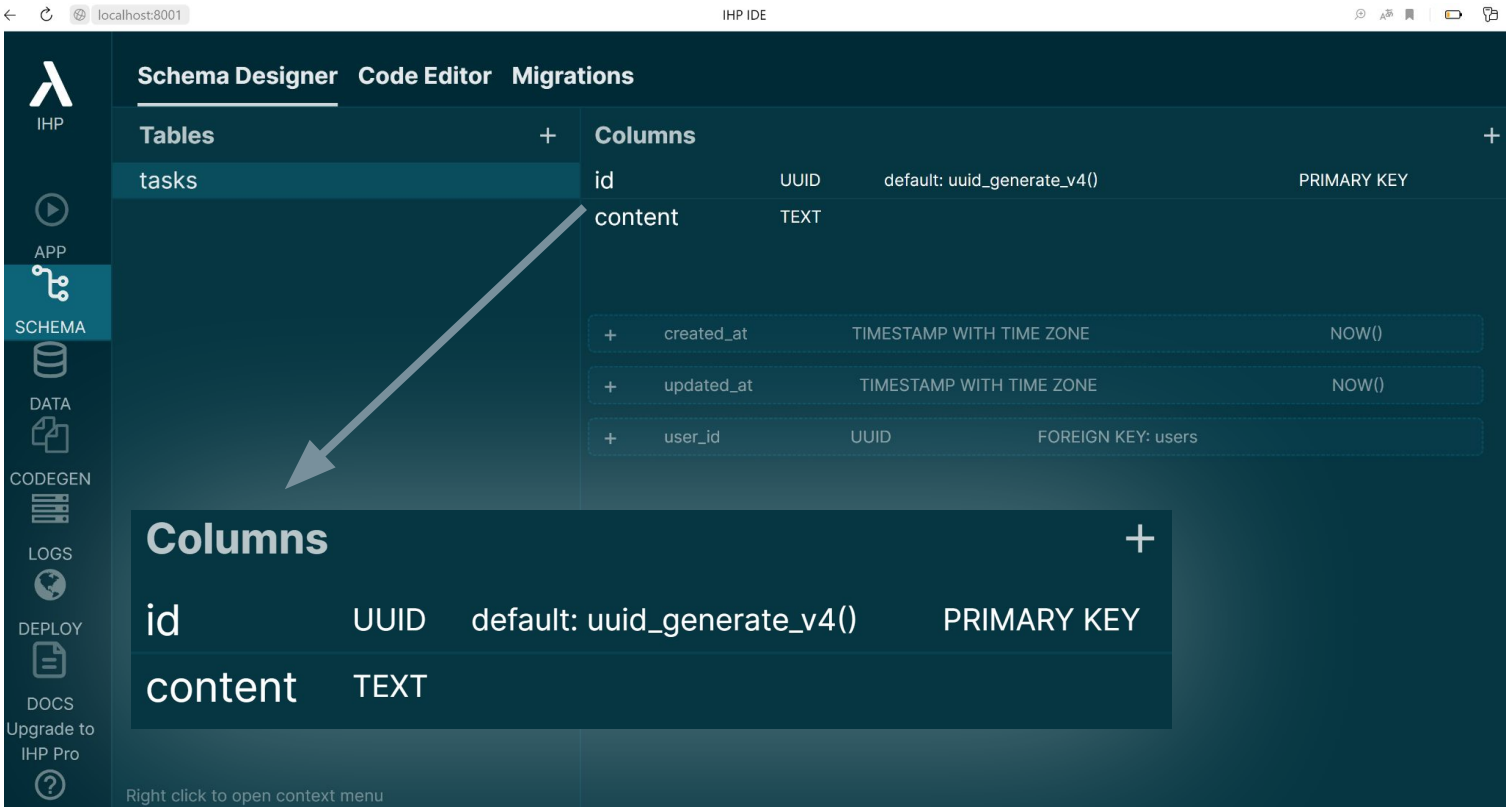
TODO-list by IHP

IHP: Database entity generation

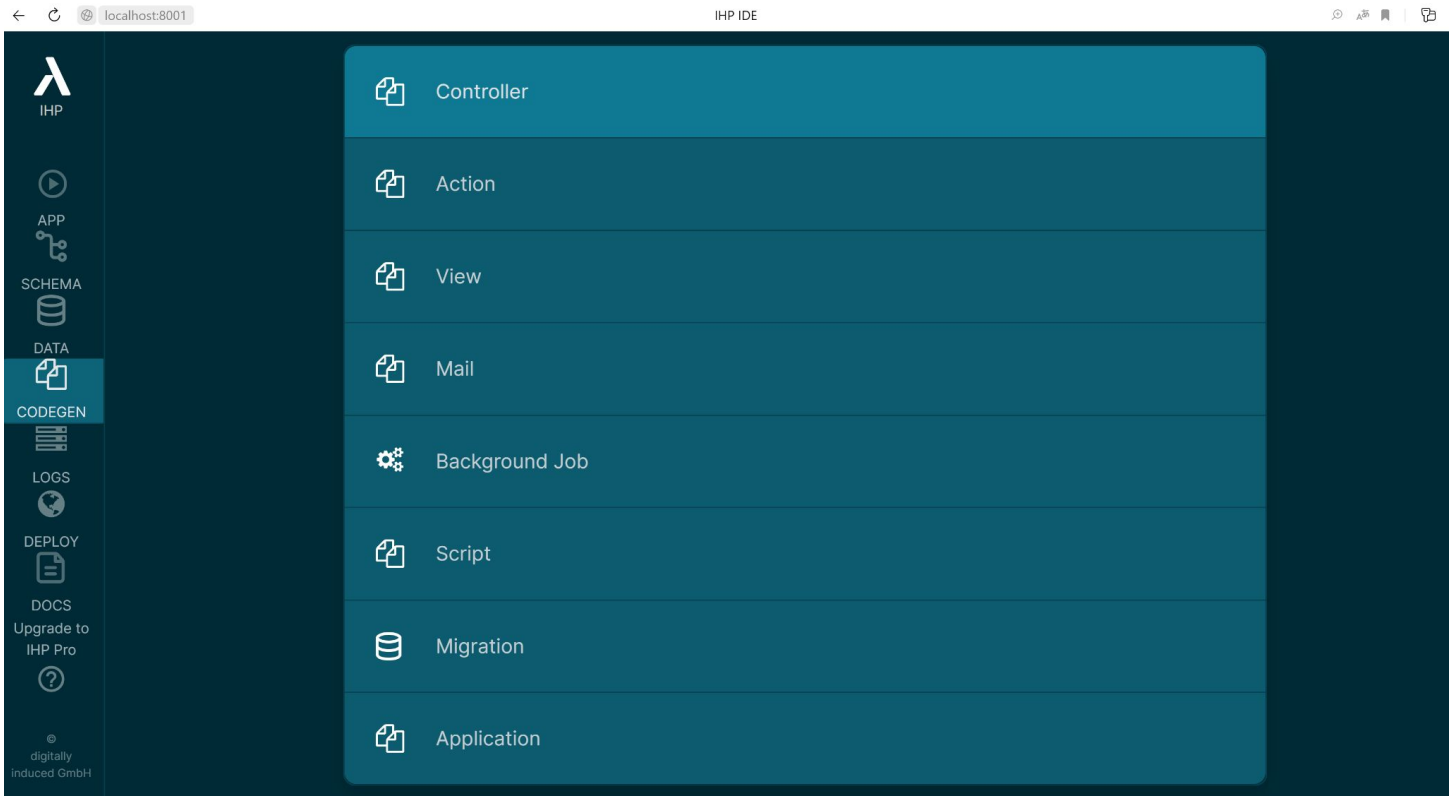




IHP: Database entity generation



IHP: Handlers generation



CODEGEN →

Controller →

Preview →

Generate

IHP: Handlers definition

```
data TasksController
    -- GET /Tasks (/api/task)
    = TasksAction

    -- POST /NewTask (/api/task)
    | NewTaskAction

    -- GET /ShowTask?taskId={id} (/api/task/{id})
    | ShowTaskAction { taskId :: !(Id Task) }
    | CreateTaskAction

    -- PUT /EditTask?taskId={id} (/api/task/{id})
    | EditTaskAction { taskId :: !(Id Task) }
    | UpdateTaskAction { taskId :: !(Id Task) }

    -- DELETE /DeleteTask?taskId={id} (/api/task/{id})
    | DeleteTaskAction { taskId :: !(Id Task) }

deriving (Eq, Show, Data)
```

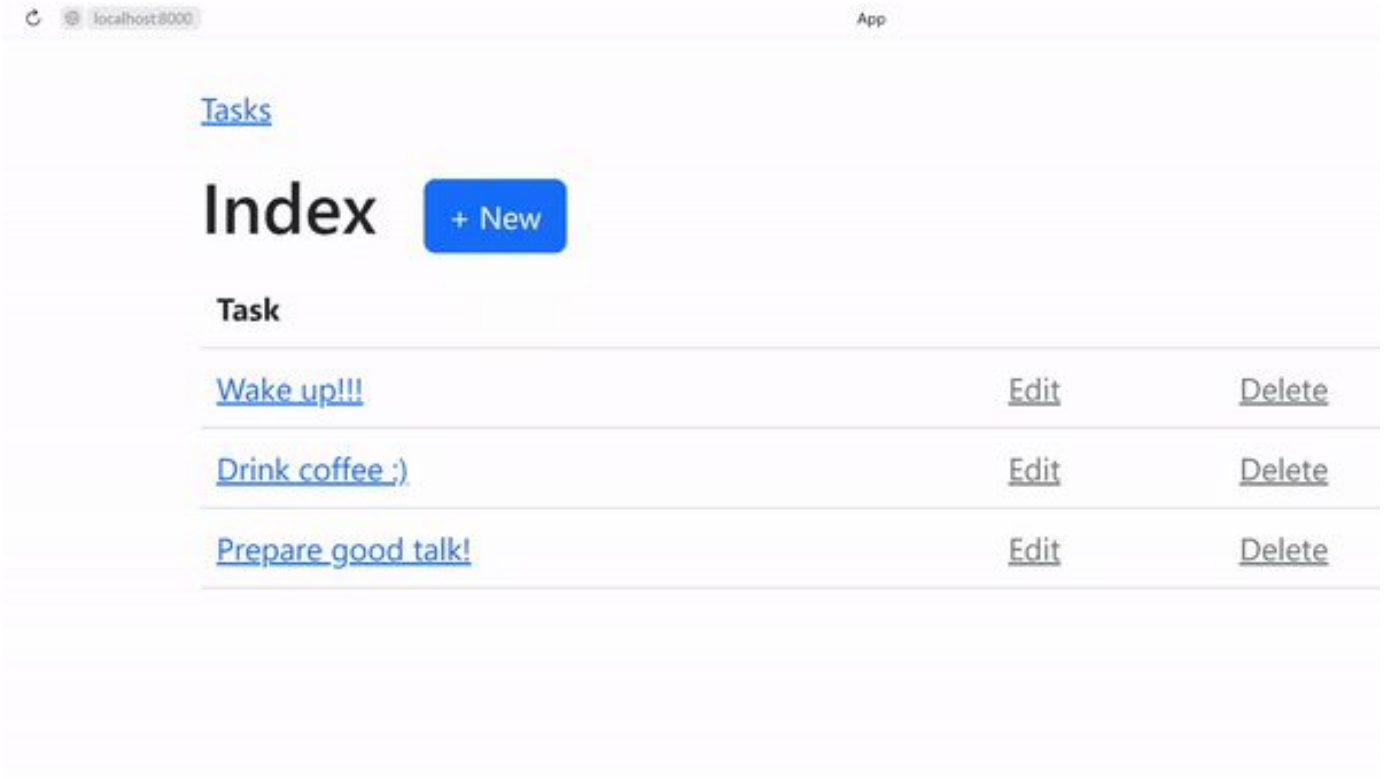
IHP: Handlers implementation

```
instance Controller TasksController where
    action TasksAction = do
        tasks <- query @Task |> fetch
        render IndexView { .. }

    action NewTaskAction = do
        let task = newRecord
        render NewView { .. }

    action ShowTaskAction { taskId } = do
        task <- fetch taskId
        render ShowView { .. }
```

IHP: Frontend

```
data ShowView = ShowView { task :: Task }

instance View ShowView where
    html ShowView { .. } = [hsx|
        {breadcrumb}
        <h1>Content</h1>
        <p>{task.content}</p>
    |]
        where
            breadcrumb = renderBreadcrumb
                            [ breadcrumbLink "Tasks" TasksAction
                            , breadcrumbText "Show Task"
                            ]
```

IHP: Ready web app



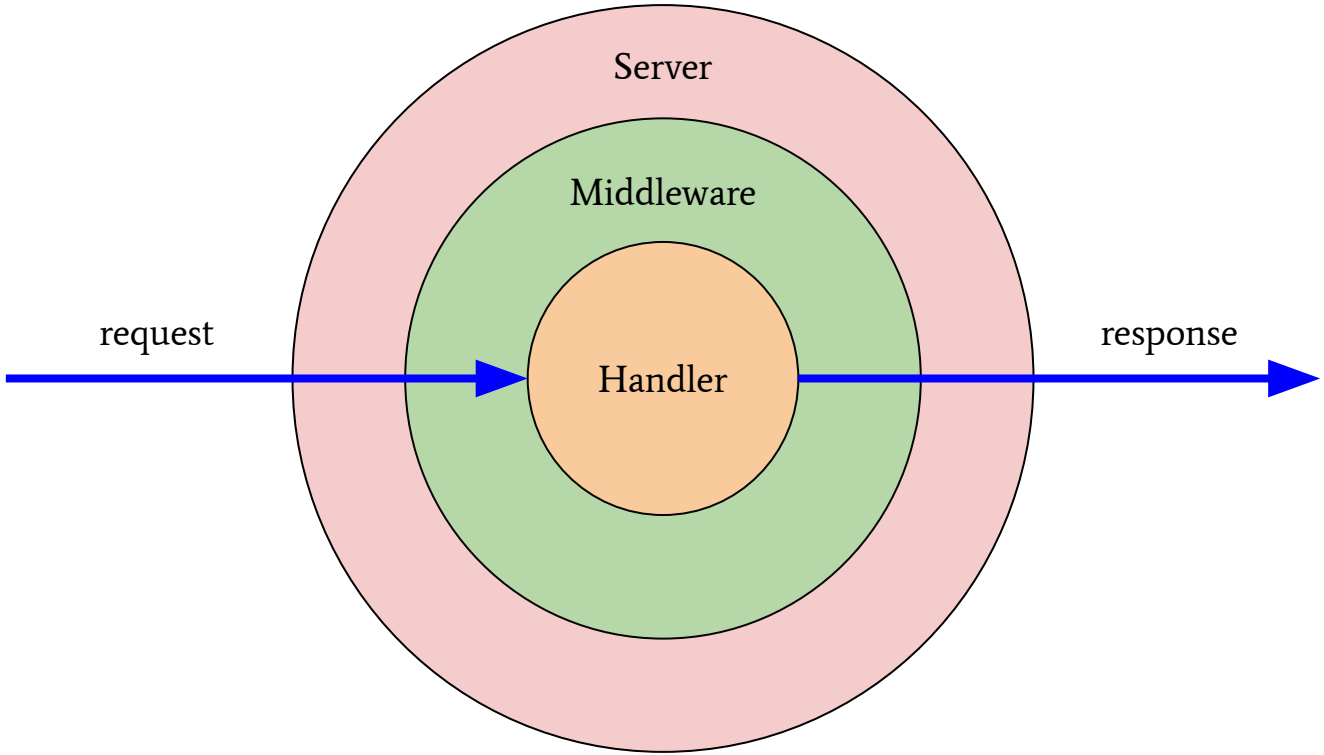
IHP: JSON handlers

(docs)

```
instance Controller TasksController where
    action TasksAction = do
        tasks <- query @Task |> fetch
        renderJson $ toJSON tasks

    action NewTaskAction = do
        let task = newRecord
        renderJson $ toJSON task

    action ShowTaskAction { taskId } = do
        task <- fetch taskId
        renderJson $ toJSON task
```

Logging middleware

Middleware



Middleware in Haskell

WAI – generic web application interface between web server and application that Haskell web tools use under the hood

---

```
type Middleware = Application -> Application

type Application = Request -> (Response -> IO ResponseReceived) -> IO ResponseReceived
```

Custom logger definition

```
logger :: Middleware

logger :: Application -> Application

logger
   :: Application
   -> Request
   -> (Response -> IO ResponseReceived)
   -> IO ResponseReceived
```

```
logger :: Middleware
logger app request responseFunc = do
  let requestedPath = intercalate "/" $
        pathInfo request
  let msg = "url-path=" <> requestedPath
  runStdoutLoggingT $ logInfoN msg
  app request responseFunc
```

Servant: Integrate logger

```
main :: IO ()
main = runStdoutLoggingT $
  withPostgresqlPool connectionStr connectionsNumber $
    \pool -> liftIO $ do
      let config = Config
         { configPool = pool
         , configPort = defaultPort
         }

      runSqlPool (runMigration migrateAll) (configPool
config)

      -- apply logger middleware
      run (configPort config) . logger $ taskListApp config
```

```
logger :: Middleware

type Middleware =
      Application -> Application
```

Yesod: Integrate logger

```
main :: IO ()
main = runStdoutLoggingT $
  withPostgresqlPool connectionStr connectionsNumber $
    \pool -> liftIO $ do
      let config = Config
         { configPool = pool
         , configPort = defaultPort
         }

      runSqlPool (runMigration migrateAll) (configPool
config)

      app <- toWaiAppPlain $ TaskList config

      -- apply logger middleware
      run (configPort config) . logger $ app
```

```
logger :: Middleware

type Middleware =
      Application -> Application
```

IHP: Integrate logger

```
config :: ConfigBuilder
config = do
    option Development
    option (AppHostname "localhost")

    -- Add custom middleware
    option $ CustomMiddleware logger
```

Middleware logs

```
[Info] url-path=api/task/5
[Info] url-path=api/task/
[Info] url-path=api/task
[Info] url-path=api/task/8/
[Info] url-path=api/task/3
[Info] url-path=/
[Info] url-path=api/task
```

Final overview

Servant: Final overview

Pros:

1. Lightweight, good when REST backend is needed as a part of a bigger app
2. Ability to use another monad for handlers and add effects you need (e.g. ReaderT)
3. Handlers could be combined to get rid of repetition
4. API type information could be used for creating type-safe handlers, and generating swagger schema, typescript types and so on

Cons:

1. Confusing to set up and run an application
2. API syntax is not clear at the first glance
3. Needs some understanding of how it works under the hood
4. Not much info in official docs/tutorials, need to look for examples at github, related articles might be outdated and have poor explanations

Yesod: Final overview

Pros:

1. Has all you need to implement simple web app, including frontend
2. Provides lots of handy, convenient functions out of the box
3. Nice and simple DSL for specifying routes
4. Easy to set up
5. Convenient and simple integration with database via Persistent library
6. Nice docs (Yesod book) with plenty of examples
7. Simple app (aka CRUD) could be implemented without deep knowledge of how the framework works under the hood
8. Some popular middleware built-in

Cons:

1. Framework, might be heavy to integrate to other projects (to use outside of its ecosystem)
2. Not so powerful API like in Servant, need to do manual conversions (from/to JSON)

IHP: Final overview

Pros:

1. Allows to make a simple app in a few minutes, using cool features like code generation and GUI schema designer
2. Built-in development server (hot reload)
3. Great docs with examples, easy-to-find info
4. No need to understand how it works to make a simple app
5. Possibility to use Elm/PureScript on top of IHP and access JavaScript/TypeScript to create a hybrid frontend
6. Good support by maintainers

Cons:

1. “No-code” GUI builder may look unfamiliar
2. High CPU usage of built-in development server
3. Relatively young, version 1.0 released at October 2022 (though, first public version was released two years ago)
4. Only supports Postgres in the open-source version, other platforms are in the roadmap for paid version
5. Not obvious how to add third-party libraries (e.g. Katip for logging)

Source code

- TODO-list Servant: github.com/alyoanton9/todo-list-servant
- TODO-list Yesod: github.com/alyoanton9/todo-list-yesod
- TODO-list IHP: github.com/alyoanton9/todo-list-ihp

That’s all!
Many thanks!