Comparing beginner Rust to JavaScript

Etienne Laurin
Septembre 2025

Words Myth

Faerie Tale

‹play in person›

Once upon a time [describe the main charater(s)]. To their unexpected surprise, [describe the conflict]. It came to pass that [describe the resolution]. They lived happily ever after.

```
nativeBuildInputs = [ bashInteractive prisma prisma-engines openssl ];

packages = [ nodejs_22 pnpm_10 ];
```

```
nativeBuildInputs = [ rustup pkg-config ];

buildInputs = [ openssl sqlite pango ];

paths = path: pkgs: lib.strings.concatStringsSep ":" (map (pkg: "${pkg}/${path}") pkgs);

bashHook = writeText "envrc" ''
  export ESC_PATH="$PATH"
  export NIX_PATH=nixpkgs=${nixpkgs}
  export PATH=${paths "bin" (localTools ++ nativeBuildInputs ++ devTools)}
  export LIBRARY_PATH=${paths "lib" buildInputs}
  export PKG_CONFIG_PATH=${paths "lib/pkgconfig" (map (p: p.dev) buildInputs)}
'';
```

```
"dependencies": {
    "@prisma/client": "5.22.0",
    "@quixo3/prisma-session-store": "^3.1.1
    "cookie-parser": "^1.4.7",
    "express": "^4.21.2",
    "express-session": "^1.18.1",
    "passport": "^0.7.0",
    "passport-anonymous": "^1.0.1",
    "passport-magic-link": "^2.1.1"
},
```

```
"devDependencies": {
    "@babel/preset-react": "^7.26.3",
    "@rollup/plugin-babel": "^6.0.4",
    "@rollup/plugin-commonjs": "^28.0.2",
    "@rollup/plugin-node-resolve": "^16.0.0",
    "@rollup/plugin-replace": "^6.0.2",
    "nodemon": "^3.1.9",
    "prisma": "5.22.0",
    "react": "^19.0.0",
    "react-dom": "^19.0.0",
    "rollup": "^4.31.0"
},
```

```
[dependencies]
async-std = "1.13.2"
chrono = "0.4.41"
dioxus = { version = "0.7.0-rc.0", features = ["router", "fullstack"] }
serde = { version = "1.0.219", features = ["derive"] }

axum = { version = "0.8.4", optional = true }
console-subscriber = { version = "0.4.1", optional = true }
diesel = { version = "2.2.12", features = [
  "sqlite", "chrono", "returning_clauses_for_sqlite_3_35", "r2d2"
], optional = true }
diesel_migrations = { version = "2.2.0", features = ["sqlite"], optional = true }
tokio = { version = "1.47.1", optional = true, features = ["tracing"] }
tower-sessions = { version = "0.14.0", optional = true }
serde_json = "1.0.143"
async-trait = "0.1.89"
```

```
const PlayerContext = createContext()
```

```
#[derive(Clone)]
struct PlayerContext {
  player: Resource<Result<Player, ServerFnError>>,
}
```

```
return <div className='content'>
            {
                name === 'play' ? <Play/>
                    : name === 'prompts' ? <Prompts/>
                    : name === 'account' ? <Account/>
                    : name === 'prompt' ? <PromptPage id={parseInt(arg)} />
                : name === 'new' ? <Game promptId={parseInt(arg)}/>
                : name === 'game' ? <Game gameId={parseInt(arg)}/>
                : <UnknownPage/>
            }
        </div>
```

```rust
#[derive(Routable, PartialEq, Clone)]
enum Route {
  #[layout(RouteLayout)]
  #[route("/play")]
  Play {},
  #[route("/prompts")]
  Prompts {},
  #[route("/account")]
  Account {},
  #[route("/new/:id")]
  NewGame { id: usize },
  #[route("/game/:id")]
  Game { id: usize },
  #[route("/:..segments")]
  NotFound { segments: Vec<String> },
}
```

```
const [player, setPlayer] = useState();
useEffect(() ⇒ {
    if(!player) getPlayer().then(setPlayer)
}, [player])
```

```
<PlayerContext.Provider value={{player, setPlayer}}>
```

```
async function getPlayer() {
    const player = await post('player')
    return player
}
```

```
let player = use_server_future(get_player)?;
use_context_provider(\/ PlayerContext { player });
```

```
function TopMenu() {
    const navigate = useContext(NavigateContext)
    return <div className='menu' style={{ justifyContent: 'end' }}>
                <Button onClick={() => navigate('play')}>play</Button>
                <Button onClick={() => navigate('prompts')}>prompts</Button>
                <Button onClick={() => navigate('account')}>account</Button>
            </div>
}
```

```
#[component]
fn TopMenu() → Element {
 rsx! {
    div {
       class: "menu",
       style: "justify-content: end",
       Link { to: Route::Play {}, "play" }
       Link { to: Route::Prompts {}, "prompts" }
       Link { to: Route::Account {}, "account" }
    }
 }
}
```

```
function UnknownPage() {
    const navigate = useContext(NavigateContext)
    useEffect(() ⇒ {
        setTimeout(() ⇒ { navigate('play') }, 1500)
    })
    return <> <span>Error</span> <Loading/> </>
}
```

```
#[component]
fn NotFound(segments: Vec<String>) → Element {
 let nav = navigator();
 use_hook(\/ {
  spawn(async move {
   async_std::task::sleep(Duration::from_secs(2))⟳;
   nav⇢push(Route::Play {});
  })
 });
 rsx! {
    Error { "Not Found" }
    Loading {}
 }
```

```
function Account() {
    const {player, setPlayer} = useContext(PlayerContext)
    return <>
                <div className='title'>Account</div>
                <div>
                    Name:{' '}
                    { player.name }
```

```
#[component]
fn Account() → Element {
 let player = use_context::<PlayerContext>()→player→suspend()?;
 match &*player→read() {
  Ok(Player{ name }) ⇒ rsx! { "Player {name}" },
  Err(error) ⇒ rsx! { Error { "{error}" } },
 }
}
```

```
use_context_provider(|| PlayerContext { player });

rsx! {
    document::Link { rel: "icon", href: FAVICON }
    document::Title { "Words Myth" }
    document::Stylesheet { href: RESET_CSS }
    document::Stylesheet { href: MAIN_CSS }
    document::Link { rel: "preconnect", href:"https://fonts.googleapis.com" }
    document::Link { rel: "preconnect", href:"https://fonts.gstatic.com", crossorigin: "" }
    document::Stylesheet { href: "https://fonts.googleapis.com/css2?family=Fuzzy+Bubbles:wght
@400;700&family=Noto+Emoji:wght@300..700&display=swap" }
    Router::<Route> {}
}
```

```
generator client {
  provider = "prisma-client-js"
  output = "../build/prisma-client"
}

datasource db {
  provider = "sqlite"
  url      = "file:./db.sqlite"
}

model Player {
  id           Int           @id @default(autoincrement())
  name         String
  participation Participant[]
}
```

```
const { PrismaClient } = prismaClient;

const prisma = new PrismaClient()
```

```
const player = async (req, res, next) ⇒ {
    if (!req.session.playerId) {
        const { id } = await prisma.player.create({ data: { name: randomName() } })
        req.session.playerId = id
    }
    req.player = await prisma.player.findUnique({
        where: { id: req.session.playerId },
    })
    next()
}
```

```
// @generated automatically by Diesel CLI

diesel::table! {
    player (id) {
        id → Integer,
        name → Text,
    }
}
```

```rust
#[derive(Queryable, Selectable)]
#[diesel(table_name = schema::player)]
#[diesel(check_for_backend(diesel::sqlite::Sqlite))]
pub struct Player { 2 implementations
    pub id: i32,
    pub name: String,
}
```

```
match session.player_id {
  None => {
    use schema::player::dsl::*;
    let p = diesel::insert_into(player)
      .values(name.eq("randomName"))
      .get_result(&mut conn)
      .map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
    Ok(RequestState { player: p, session, conn })
  }
  Some(pid) => {
    use schema::player::dsl::*;
    let res = player.filter(id.eq(pid)).select(model::Player::as_select()).first(&mut conn);
    match res {
      Ok(p) => Ok(RequestState { player: p, session, conn }),
      Err(_) => {
        session
          .replace(SessionState::<()> { player_id: None, session: () })
          ⟳
          .map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
        Err(StatusCode::INTERNAL_SERVER_ERROR)
      }
    }
  }
```

```rust
fn create_connection_pool() -> Pool<ConnectionManager<diesel::SqliteConnection>> {
  let database_url = env::var("DATABASE_URL").expect("DATABASE_URL must be set");
  let manager = ConnectionManager::<SqliteConnection>::new(database_url);
  Pool::builder().test_on_check_out(true).build(manager)
    .expect("Could not build connection pool")
}
```

```rust
#[derive(Clone)]
pub struct ServerState {
  db_pool: Pool<ConnectionManager<diesel::SqliteConnection>>,
}
```

```
pub fn get_connection(
  extension: Extension<ServerState>,
) → PooledConnection<ConnectionManager<SqliteConnection>> {
  extension⇢0⇢db_pool⇢get()?
}
```

```rust
#[async_trait]
impl SessionStore for DbSessionStore {
  #[allow(deprecated)]
  async fn save(&self, record: &session::Record) -> session_store::Result<()> {
    let res = diesel::insert_into(schema::session::dsl::session)
      .values(model::Session {
        id: format!("{:x}", record.id.0),
        data: serde_json::to_string(&record.data)
          .map_err(|err| session_store::Error::Encode(format!("{}", err)))?,
        expiresat: chrono::NaiveDateTime::from_timestamp(record.expiry_date.unix_timestamp(), 0
),
      })
      .execute(&mut self.db_pool.get()?);
    match res {
      Ok(_) => Ok(()),
      Err(err) => Err(session_store::Error::Backend(format!("{}", err))),
    }
  }
```

```rust
#[derive(Default, Deserialize, Serialize)]
pub struct SessionState<T> {
  pub player_id: Option<i32>,
  session: T,
}

impl SessionState<Session> {
  async fn replace(&mut self, other: SessionState<()>) → Result<(), session::Error> {
    self→session→insert(PLAYER_SESSION_KEY, other)⟳
  }
}

pub async fn get_session(session: Session) → SessionState<Session> {
  let SessionState::<()> { player_id, ⋯ } =
    session→get(PLAYER_SESSION_KEY)⟳?→unwrap_or_default();
  SessionState::<Session> { session, player_id }
}
```

```rust
pub struct RequestState {
  pub conn: PooledConnection<ConnectionManager<SqliteConnection>>,
  pub session: SessionState<Session>,
  pub player: model::Player,
}

impl<S> FromRequestParts<S> for RequestState
where
  S: Send + Sync,
{
  type Rejection = StatusCode;
  async fn from_request_parts(parts: &mut Parts, _state: &S) → Result<Self, Self::Rejection> {
    use axum::RequestPartsExt;
    let mut session = get_session(
      parts
        →extract::<Session>()
        ⟳
        →map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?,
    )
    ⟳;
    let mut conn = get_connection(
      parts
        →extract::<Extension<ServerState>>()
        ⟳
        →map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?,
    );
    match session→player_id {
      None ⇒ {
        use schema::player::dsl::*;
        let p = diesel::insert_into(player)
          →values(name→eq("randomName"))
          →get_result(&mut conn)
          →map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
        Ok(RequestState { player: p, session, conn })
      }
      Some(pid) ⇒ {
        use schema::player::dsl::*;
        let res = player→filter(id→eq(pid))→select(model::Player::as_select())→first(&mut conn);
        match res {
          Ok(p) ⇒ Ok(RequestState { player: p, session, conn }),
          Err(_) ⇒ {
            session
              →replace(SessionState::<()> { player_id: None, session: () })
              ⟳
              →map_err(|_| StatusCode::INTERNAL_SERVER_ERROR)?;
            Err(StatusCode::INTERNAL_SERVER_ERROR)
          }
        }
      }
    }
  }
```

```rust
#[derive(Deserialize, Serialize, Clone)]
pub struct Player {
  name: String,
}

#[server]
pub async fn get_player() → Result<Player, ServerFnError> {
  let req: RequestState =
    extract()⟳→map_err(|e| ServerFnError::ServerError(format!("{}", e)))?;
  Ok(Player { name: req→player→name })
}
```

JavaScript

- React components and hooks
- Single-threaded
- Throw anything
- Even with TS, duck types
- GC
- Slow runtime
- JSON, YAML, .config.js

Rust

- Dioxus components and hooks
- Multi-threaded
- Exact error handling
- Strict typing
- Borrow checker
- Slow builds
- TOML everywhere

Caveats

- Picked Dioxus and Diesel arbitrarily, seemed popular and beginner-friendly

Questions?

atnnn.com