🚀 XMTP Agent SDK

Build event-driven messaging agents on XMTP

`xmtp.org/miniapps`

What is XMTP Agent SDK?

Build **conversational agents** that:

- 📨 Respond to messages automatically
- 🎯 Handle events (DMs, groups, reactions, etc.)
- 🔌 Use middleware for extensibility
- 💾 Persist conversations in local database

Think **Express.js for messaging** 🎉

Installation

Choose your package manager:

```
npm install @xmtp/agent-sdk

# or
pnpm add @xmtp/agent-sdk

# or
yarn add @xmtp/agent-sdk
```

📋 Step 1: Setup Environment Variables

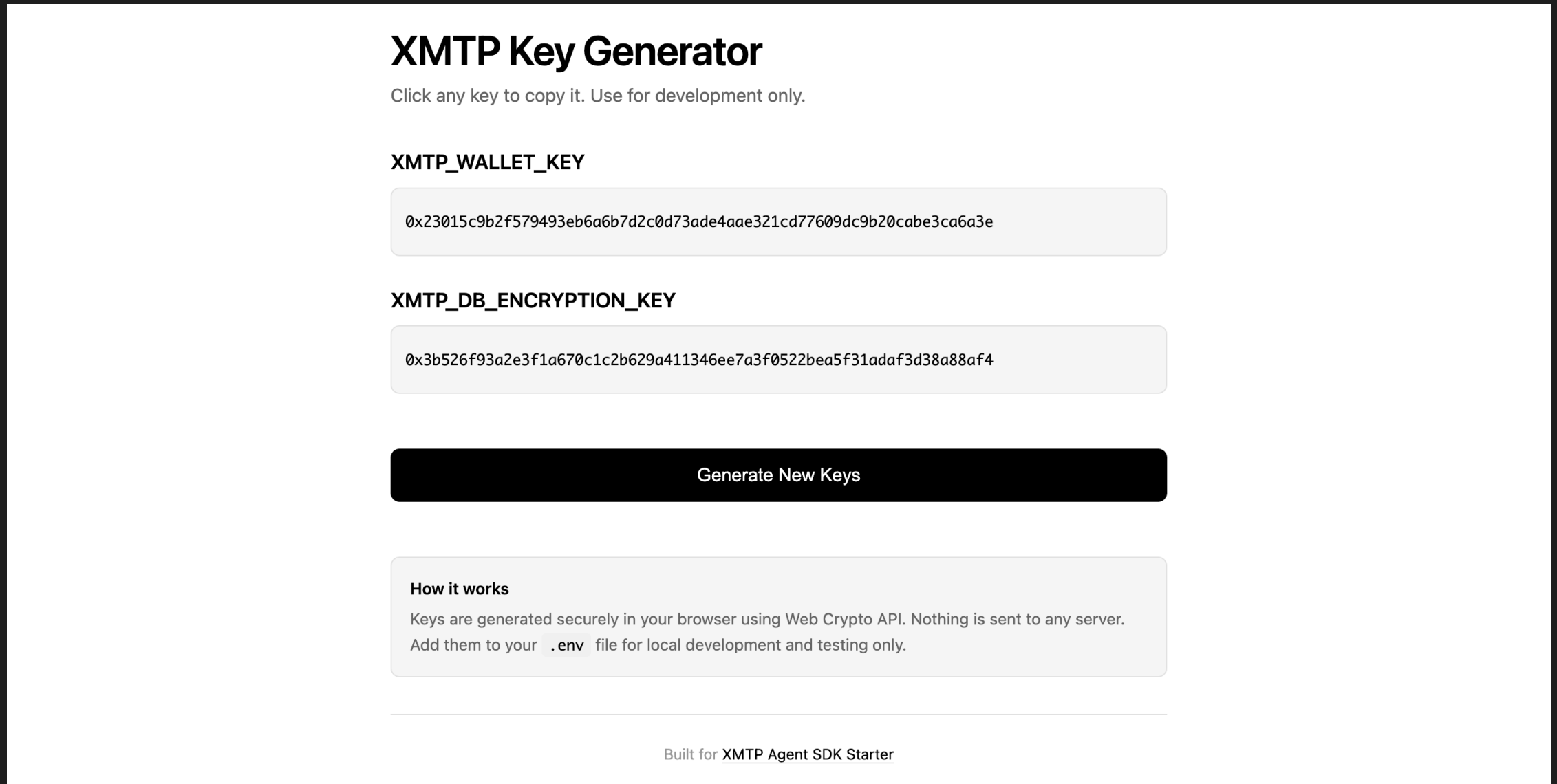
Environment Variables

Create a `.env` file in your project root:

```
# Required: Your wallet private key (hex format)
XMTP_WALLET_KEY=0x1234567890abcdef...

# Optional: Network environment
XMTP_ENV=dev  # or 'production'

# Optional: Database persistence
XMTP_DB_DIRECTORY=./data

# Optional: Encrypt your database
XMTP_DB_ENCRYPTION_KEY=0xabcd...1234
```

💡 Tip: Generate test keys at xmtp.github.io/agent-sdk-starter

XMTP Key Generator



Environment Variables Explained

Variable	Purpose
`XMTP_WALLET_KEY`	Private key for your agent's wallet identity
`XMTP_ENV`	Network (`dev` or `production`)
`XMTP_DB_DIRECTORY`	Where to store conversation data
`XMTP_DB_ENCRYPTION_KEY`	Secure your local database

⚠️ **Important**: Keep `XMTP_WALLET_KEY` and `XMTP_DB_ENCRYPTION_KEY` secret!

🎬 Step 2: Create Your Agent

Create Agent from Environment

Load your `.env` file and create the agent:

```
import { Agent } from "@xmtp/agent-sdk";
import { loadEnvFile } from "node:process";

// Load environment variables
loadEnvFile(".env");

// Create agent using environment variables
const agent = await Agent.createFromEnv();
```

That's it! Your agent is ready 🎉

Alternative: Manual Setup

If you prefer not to use environment variables:

```
import { Agent } from "@xmtp/agent-sdk";
import { createUser, createSigner } from "@xmtp/agent-sdk/user";

const user = createUser();
const signer = createSigner(user);

const agent = await Agent.create(signer, {
  env: "dev",
  dbPath: null, // in-memory (or provide path)
});
```

💬 Step 3: Responding to Messages

Listen to Text Messages

```
agent.on("text", async (ctx) => {
  console.log(`Received: ${ctx.message.content}`);

  // Send a reply
  await ctx.sendText("Hello! 👋");
});
```

Simple event-driven pattern you already know!

Available Message Events

```
agent.on("text", async (ctx) => {
  /* text messages */
});
agent.on("reaction", async (ctx) => {
  /* emoji reactions */
});
agent.on("reply", async (ctx) => {
  /* message replies */
});
agent.on("attachment", async (ctx) => {
  /* file attachments */
});
agent.on("markdown", async (ctx) => {
  /* markdown content */
});
agent.on("read-receipt", async (ctx) => {
  /* read receipts */
});
agent.on("group-update", async (ctx) => {
  /* group changes */
});
```

Rich Context Helper Methods

Every event handler receives a `ctx` with helpful methods:

```
agent.on("text", async (ctx) => {
  // Simple text reply
  await ctx.sendText("Hi there!");

  // Reply to the message
  await ctx.sendTextReply("Replying to you!");

  // Send markdown
  await ctx.sendMarkdown("**Bold** and *italic*");

  // React to message
  await ctx.sendReaction("👍");
});
```

Example: Echo Bot

```
agent.on("text", async (ctx) => {
  const message = ctx.message.content;

  // Echo back the message
  await ctx.sendText(`You said: ${message}`);
});

agent.on("start", () => {
  console.log("Echo bot is online! 🎙");
});

await agent.start();
```

👥 Step 4: Creating Conversations

Start a Direct Message (DM)

```
// Option 1: Use raw Ethereum address
const dm1 = await agent.createDmWithAddress(
  "0x1234567890123456789012345678901234567890",
);

// Option 2: Resolve ENS/web3 names with convenience helper
import { createNameResolver } from "@xmtp/agent-sdk/user";
const resolver = createNameResolver("your-web3bio-api-key");

const address = await resolver("vitalik.eth");
const dm2 = await agent.createDmWithAddress(address);

// Send a message
await dm2.send("Hello! This is a DM.");
```

Perfect for reaching out to users!

Create a Group Conversation

```
// Create a group with multiple addresses
const group = await agent.createGroupWithAddresses([
  "0x1111111111111111111111111111111111111111",
  "0x2222222222222222222222222222222222222222",
  "0x3333333333333333333333333333333333333333",
]);

// Send to the group
await group.send("Welcome to the group! 🎉");

// Add more members
await group.addMembers(["0x4444444444444444444444444444444444444444"]);
```

Listen for New Conversations

```
// Listen for all new conversations
agent.on("conversation", async (ctx) => {
  console.log(`New conversation: ${ctx.conversation.id}`);
});

// Listen specifically for DMs
agent.on("dm", async (ctx) => {
  await ctx.sendText("Thanks for the DM! 💌");
});

// Listen specifically for groups
agent.on("group", async (ctx) => {
  await ctx.sendMarkdown("**Hello, group!** 👋");
});
```

🔌 Step 5: Adding Middleware

What is Middleware?

Middleware lets you:

- ✅ Filter messages
- ✅ Add logging/analytics
- ✅ Implement rate limiting
- ✅ Route commands
- ✅ Transform messages

Just like Express.js middleware!

Simple Middleware Example

```
import type { AgentMiddleware } from "@xmtp/agent-sdk";

const logger: AgentMiddleware = async (ctx, next) => {
  console.log(`📨 Message from: ${ctx.message.senderInboxId}`);
  console.log(`📝 Content: ${ctx.message.content}`);

  // Continue to next middleware
  await next();
};

agent.use(logger);
```

Filter Messages with Middleware

```
import { filter } from "@xmtp/agent-sdk";
import type { AgentMiddleware } from "@xmtp/agent-sdk";

const onlyText: AgentMiddleware = async (ctx, next) => {
  // Only process text messages
  if (filter.isText(ctx.message)) {
    await next();
  }
  // Otherwise, stop the chain (return)
};

agent.use(onlyText);
```

Built-in CommandRouter Middleware

```
import { CommandRouter } from "@xmtp/agent-sdk/middleware";

const router = new CommandRouter();

router.command("/hello", async (ctx) => {
  await ctx.sendText("Hi there! 👋");
});

router.command("/help", async (ctx) => {
  await ctx.sendText("Available commands: /hello, /help, /status");
});

router.default(async (ctx) => {
  await ctx.sendText(`Unknown command: ${ctx.message.content}`);
});

agent.use(router.middleware());
```

Multiple Middleware Chain

```
const loggerMW: AgentMiddleware = async (ctx, next) => {
  console.log("📨 Message received");
  await next();
};

const filterSelfMW: AgentMiddleware = async (ctx, next) => {
  if (!filter.fromSelf(ctx.message, ctx.client)) {
    await next();
  }
};

const rateLimitMW: AgentMiddleware = async (ctx, next) => {
  // Your rate limiting logic
  await next();
};

// Add all middleware
agent.use(loggerMW, filterSelfMW, rateLimitMW);
```

Best Practices

1. 🔐 **Keep secrets safe**: Use `.env` for keys, never commit them
2. 💾 **Use database**: Set `XMTP_DB_DIRECTORY` for persistence
3. 🛡️ **Filter wisely**: Avoid infinite loops, filter self-messages
4. 🔌 **Compose middleware**: Keep middleware focused and reusable
5. 🐛 **Handle errors**: Add error middleware for graceful failures

Resources

📚 **Documentation**: docs.xmtp.org/agents

📦 **NPM Package**: npm.im/@xmtp/agent-sdk

💻 **GitHub**: github.com/xmtp/xmtp-js

👷 **Starter Kit**: github.com/xmtp/agent-sdk-starter