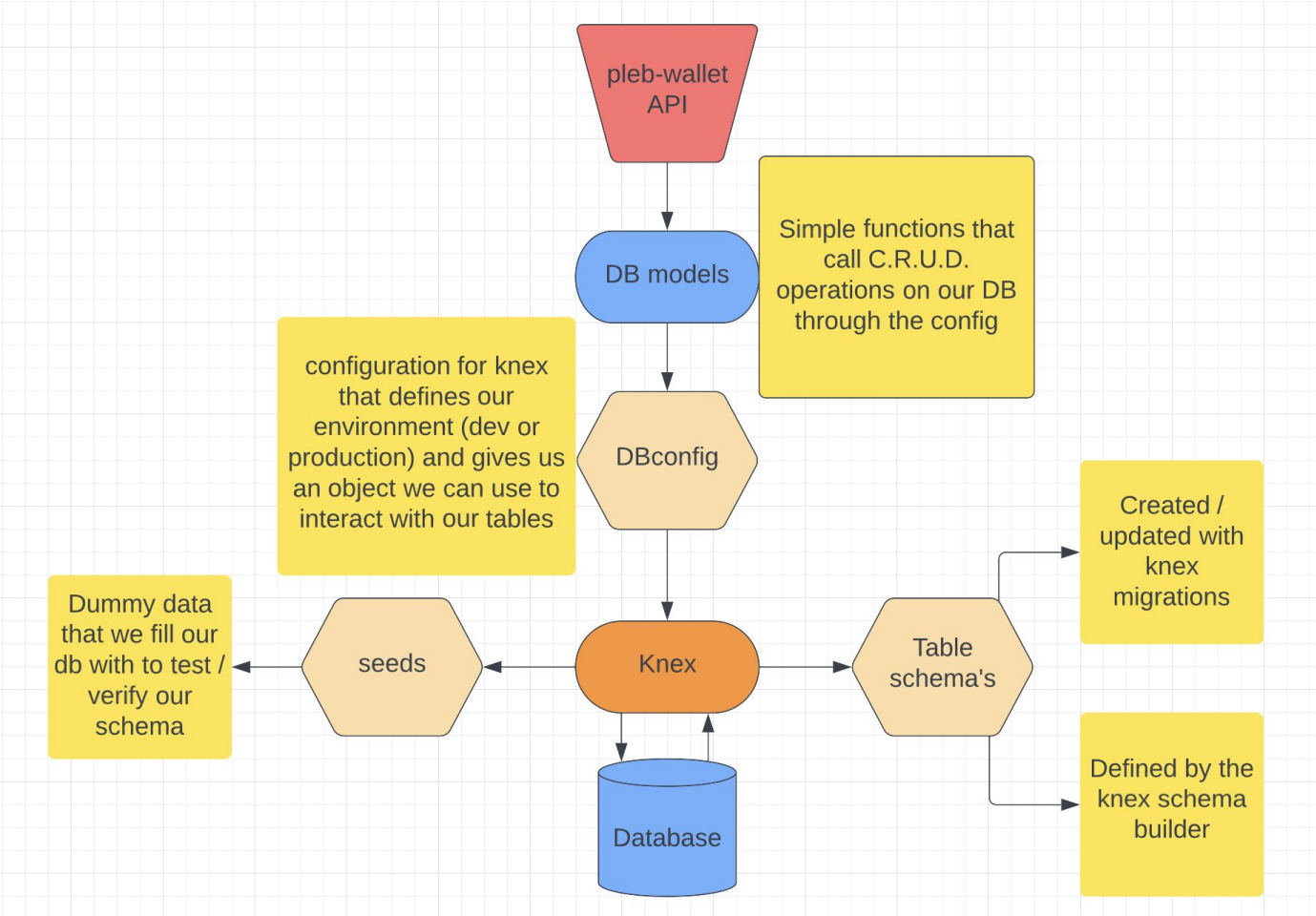
Lesson 11: Connecting the API and Database

Create db models (functions that interact with our database) for our API to use to talk to the database

Full knex setup from a high level



What are db models in knex?

Database models in Knex.js are the defined structures and helper functions that you will use to interact with your database. Essentially, they provide an interface for querying and manipulating the data stored in a specific table of your database.

Here's how to think about it:

Data Representation: Each model usually represents a particular table in your database. For example, a `Users` model would represent a `users` table.

Querying and Manipulating Data: Models provide methods for retrieving and changing data in your database. For example, a `findAllUsers` function on your `Users` model would use the Knex SDK to retrieve all users from your `users` table.

Knex queries

Knex provides a straightforward way to build SQL queries with JavaScript.

The knex cheatsheet has some common examples of SELECT queries and WHERE clauses in Knex.

```
knex
  .from('books')
  .select('title', 'author', 'year')
```

Where

```
  .where('title', 'Hello')
  .where({ title: 'Hello' })
  .whereIn('id', [1, 2, 3])
  .whereNot(...)
  .whereNotIn('id', [1, 2, 3])
```

Where conditions

```
  .whereNull('updated_at')
  .whereNotNull(...)
```

```
  .whereExists('updated_at')
  .whereNotExists(...)
```

```
  .whereBetween('votes', [1, 100])
  .whereNotBetween(...)
```

```
  .whereRaw('id = ?', [1])
```

Where grouping

```
  .where(function () {
    this
      .where('id', 1)
      .orWhere('id', '>', 10)
  })
```

Getting started on our knex queries

- Add a ‘models’ directory inside of your ‘db’ folder
  - Now your db folder should look like this:

- Create a new file called user.js
  - Import db from our dbConfig at the top of the file:

```
1    const db = require("../dbConfig");
```

Remember dbConfig?

dbConfig imports the knex library, retrieves the appropriate database configuration from the knexfile based on the current environment (defaulting to "development"), and initializes a knex instance with this configuration. The initialized instance db is then exported to be used in other files, particularly the models, to interact with the database.

```
const knex = require("knex");

const config = require("../knexfile");

const env = process.env.NODE_ENV || "development";

const db = knex(config[env]);

module.exports = db;
```

Creating our user models in user.js

```
// First, we require our configured instance of knex from the dbConfig.js file.
const db = require("../dbConfig");

// We then export an object with several methods, each representing a different database operation
module.exports = {
 // The findAll method retrieves all records from the 'users' table
 findAll: () => {
   return db("users");
 },
 // The findByUsername method retrieves the first record in the 'users' table where the username matches the provided username
 findByUsername: (username) => {
   return db("users").where({ username }).first();
 },
 // The create method inserts a new record (the 'user' object) into the 'users' table and returns the newly created user
 create: (user) => {
   return db("users").insert(user).returning("*");
 },
 // The update method finds a user in the 'users' table with the matching id and updates their record with the new data contained in the
'user' object. It then returns the updated user
 update: (id, user) => {
   return db("users").where({ id }).update(user).returning("*");
 },
 // The delete method finds a user in the 'users' table with the matching id and removes their record from the table
 delete: (id) => {
   return db("users").where({ id }).del();
 },
};
```

Update getUsers endpoint in routers/usersRouter

```
// add imports for the user model and our two auth middlewares
const User = require("../db/models/user");
const authenticate = require("./middleware/authenticate.js");
const authenticateAdmin = require("./middleware/authenticateAdmin.js");

// GET all users
// Before processing the request, we apply the 'authenticateAdmin' middleware to protect the ability to see all users
router.get("/", authenticateAdmin, (req, res) => {
 // Call the 'findAll' method from our User model. This method retrieves all user records from the database.
 User.findAll()
   .then((users) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200 (OK)
     // and the list of users retrieved from the database.
     res.status(200).json(users);
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500 (Internal Server
Error)
     // and the error that occurred. This might be due to a database issue, a network issue, etc.
     res.status(500).json(err);
   });
});
```

Update getUserByUsername endpoint

```
// GET user by their username
// Using 'authenticate' middleware to verify the client's authentication token.
router.get("/user", authenticate, async (req, res) => {
 // Get the JWT (JSON Web Token) from the 'authorization' header of the request.
 const token = req.headers.authorization;
 // Retrieve the secret key for JWT verification from environment variables.
 const secret = process.env.JWT_SECRET;

 // Use the 'verify' method from the 'jsonwebtoken' library to decode the token.
 jwt.verify(token, secret, (err, decodedToken) => {
   // If an error occurred during token decoding (perhaps because the token is invalid or the secret is incorrect),
   // respond with a status of 401 (Unauthorized) and a message about the error.
   if (err) {
     res.status(401).json({ message: "Error decoding token", Error: err });
   }

   // If the token was successfully decoded, find a user with a username that matches the username in the decoded token.
   User.findByUsername(decodedToken.username)
     .then((user) => {
       // If the promise resolves (i.e., the user was found), respond with a status of 200 (OK) and the user's data.
       console.log(user);
       res.status(200).json(user);
     })
     .catch((err) => {
       // If the promise is rejected (i.e., an error occurred), respond with a status of 500 (Internal Server Error) and the error.
       res.status(500).json(err);
     });
 });
});
```

Update register endpoint

```
// POST a user to register
router.post("/register", (req, res) => {
 // We are using the bcrypt library to hash the password provided in the request body.
 // This enhances security by ensuring that the plain text password isn't stored directly in the database.
 // The '14' here is the cost factor that determines the complexity of the hashing process.
 const hash = bcrypt.hashSync(req.body.password, 14);

 // We then replace the plain text password in the request body with the hashed password.
 req.body.password = hash;

 // The updated request body (which now includes the hashed password) is passed to the 'add' method from our User model.
 // This method will create a new user record in the database.
 User.create(req.body)
   .then((user) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200 (OK) and the newly created user.
     res.status(200).json({ data: user });
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500 (Internal Server Error) and the error
that occurred.
     res.status(500).json({ error: err });
   });
});
```

Reviewing our login endpoint

```
// POST a user to login
router.post("/login", (req, res) => {
 // Extract the username and password from the request body
 const { username, password } = req.body;

 // Placeholder user object - later we will fetch the real user from the database
 const DBuser = {
   username: "test",
   password: "pass1",
 };
 // Hash the password from the request body using bcrypt
 // Later we will compare this hash to the hash stored in the database
 // But for now, we will just do it manually
 const hashedPassword = bcrypt.hashSync(DBuser.password, 14);

 // Check if the user exists and the password matches using bcrypt
 if (DBuser && bcrypt.compareSync(password, hashedPassword)) {
   // Generate a JSON Web Token (JWT) for the user
   const token = generateToken(DBuser);

   // Send a success response with the JWT and user data
   res
     .status(200)
     .json({ message: `Welcome ${DBuser.username}!`, token, DBuser });
 } else {
   // Send an error response if the credentials are invalid
   res.status(401).json({ message: "Invalid credentials" });
 }
});
```

Update login endpoint

```
// POST a user to login
router.post("/login", (req, res) => {
 // Extract the 'username' and 'password' fields from the request body. These are provided by the client when they make the request.
 const { username, password } = req.body;

 // Call the 'findByUsername' method from our User model. This method retrieves the first user record from the database that matches the provided username.
 User.findByUsername(username)
   .then((user) => {
     // Check if a user was found and if the provided password, when hashed, matches the hashed password stored in the database.
     if (user && bcrypt.compareSync(password, user.password)) {
       // If both conditions are met, the login is successful. We then generate a token for the user. This token will be used for subsequent authenticated requests.
       const token = generateToken(user);
       // Respond with a status of 200 (OK), a welcome message, the generated token, and user information.
       res
         .status(200)
         .json({ message: `Welcome ${user.username}!`, token, user });
     } else {
       // If the user wasn't found or the password doesn't match, respond with a status of 401 (Unauthorized) and a message indicating the credentials were invalid.
       res.status(401).json({ message: "Invalid credentials"});
     }
   })
   .catch((err) => {
     // If an error occurs during the process, log the error and respond with a status of 500 (Internal Server Error) and the error.
     console.log(err);
     res.status(500).json({ error: err });
   });
});
```

Update the update user endpoint

```
// PUT a user to update them
// The 'authenticateAdmin' middleware function is used to ensure that only the admin is authorized to update the
user.
router.put("/:id", authenticateAdmin, (req, res) => {
 // Call the 'update' method from our User model with the provided id and body of the request.
 // The 'update' method will update the user record in the database that matches the provided id with the data in the
request body.
 User.update(req.params.id, req.body)
   .then((user) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200
(OK)
     // and the updated user record from the database.
     res.status(200).json(user);
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500
(Internal Server Error)
     // and the error that occurred. This might be due to a database issue, a network issue, etc.
     res.status(500).json(err);
   });
});
```

Update the delete user endpoint

```
// DELETE a user
router.delete("/:id", authenticateAdmin, (req, res) => {
 // Before running the delete operation, the `authenticateAdmin` middleware function is run.
 // This function checks whether the user making the request has the appropriate admin privileges.

 // Then we call the 'delete' method from our User model, passing in the user id extracted from the route parameters.
 User.delete(req.params.id)
   .then((user) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200
(OK)
     // and the user that was deleted from the database.
     res.status(200).json(user);
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500
(Internal Server Error)
     // and the error that occurred. This might be due to a database issue, a network issue, etc.
     res.status(500).json(err);
   });
});
```

Update authenticateAdmin middleware

```
const jwt = require("jsonwebtoken");
const User = require("../../db/models/user");

module.exports = (req, res, next) => {
 // Extracting the token from the request header
 const token = req.headers.authorization;

 // Setting up the JWT secret for token verification
 const secret = process.env.JWT_SECRET || "Satoshi Nakamoto";
 // Setting the admin key for admin verification
 const key = process.env.ADMIN_KEY || "1234";

 // If token is present, attempt to verify it using the JWT module
 if (token) {
   jwt.verify(token, secret, async (err, decodedToken) => {
     // If token is not verified, return 401 error
     if (err || !decodedToken) {
       res.status(401).json({ message: "Error with your verification" });
     } else {
       // If token is verified, find the user using their username from the database
       const user = await User.findByUsername(decodedToken.username);

       // Extracting admin key from user object if it exists
       const adminKey = user?.adminKey?.toString() ?? "";
       // Checking if extracted admin key matches with the one in env variables
       if (adminKey !== key) {
         // If admin key does not match, return 401 error
         res.status(401).json({ message: "Must be an admin" });
       } else {
         // If admin key matches, let the endpoint continue executing
         next();
       }
     }
   });
 } else {
   // If no token is present, return 401 error
   res.status(401).json({ message: "No token!" });
 }
};
```

Test the updated /users endpoints

With Postman

Flow for testing all of the /users endpoints

- Call POST /users/register with a new user object containing username, password, and adminKey set to “1234”
- Now call POST /users/login with your new user’s username and password.
- Copy the JWT that’s generated on login
- Add the JWT to an “authorization” header in a call to GET /users/user and see that you can get your newly created user
- Add the JWT to an “authorization” header in a call to PUT /users/:id with a body that updates the username
- Login again since you updated your username
- Take the token and add it to the GET /users header call it to get all users and see that your username has been updated
- Finally add the token to the DELETE /users/:id and check that the user was successfully deleted

Now we can create a similar db model for invoices

First add an invoice.js file inside of db/models directory

db/models/invoice.js

```
// First, we require our configured instance of knex from the dbConfig.js file.
const db = require("../dbConfig");

module.exports = {
 // The findAll method retrieves all records from the 'invoices' table
 findAll: () => {
   return db("invoices");
 },
 // The findOne method retrieves the first record in the 'invoices' table where the payment_request matches the provided payment_request
 findOne: (payment_request) => {
   return db("invoices").where({ payment_request }).first();
 },
 // The create method inserts a new record (the 'invoice' object) into the 'invoices' table and returns the newly created invoice
 create: (invoice) => {
   return db("invoices").insert(invoice).returning("*");
 },
 // The update method finds an invoice in the 'invoices' table with the matching payment_request and updates their record with the new data
contained in the 'invoice' object. It then returns the updated invoice
 update: (payment_request, invoice) => {
   return db("invoices")
     .where({ payment_request })
     .update(invoice)
     .returning("*");
 },
 // The delete method finds an invoice in the 'invoices' table with the matching id and removes their record from the table
 delete: (id) => {
   return db("invoices").where({ id }).del();
 },
};
```

Update get all invoices endpoint

```
const Invoice = require("../db/models/invoice.js");

• • •

// GET all invoices from the database
router.get("/invoices", (req, res) => {
 // Call the 'findAll' method from our Invoice model. This method retrieves all invoice records from the database.
 Invoice.findAll()
   .then((invoices) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200
(OK)
     // and the list of invoices retrieved from the database.
     res.status(200).json(invoices);
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500
(Internal Server Error)
     // and the error that occurred. This might be due to a database issue, a network issue, etc.
     res.status(500).json(err);
   });
});
```

Update create invoice endpoint

```
// POST required info to create an invoice
router.post("/invoice", authenticate, (req, res) => {
 // The 'authenticate' middleware function is called before the main handler.
 // This function checks if the user making the request is authenticated.

 // Extract 'value', 'memo', and 'user_id' properties from the body of the incoming request.
 const { value, memo, user_id } = req.body;
 // Call the 'createInvoice' function, passing the extracted properties as an object.
 // This function creates a new invoice record in the database.
 createInvoice({ value, memo, user_id })
   .then((invoice) => {
     // If the promise resolves (i.e., the operation was successful), we send back a response with a status of 200 (OK)
     // and the invoice object retrieved from the database.
     res.status(200).json(invoice);
   })
   .catch((err) => {
     // If the promise is rejected (i.e., the operation fails), we send back a response with a status of 500 (Internal Server
Error)
     // and the error that occurred. This could be due to a database issue, a network issue, etc.
     res.status(500).json(err);
   });
});
```

Update createInvoice function in lnd.js

```
const createInvoice = async ({ value, memo, user_id }) => {
 // Use the 'addInvoice' method from the Lightning service of the 'grpc' module to create an invoice.
 // This method requires an object parameter with 'value' and 'memo' properties.
 // This method is asynchronous, so we use 'await' to pause execution until it completes.
 const invoice = await lnd.services.Lightning.addInvoice({
   value: value,
   memo: memo,
 });

 // After creating the invoice with the Lightning service, we create a record in our own database using the 'Invoice' model's 'create' method.
 // This method requires an object parameter with properties for 'payment_request', 'value', 'memo', 'settled', 'send', and 'user_id'.
 // Note that 'settled' is set to false (since the invoice has just been created and is not yet paid), and 'send' is also false (since we haven't
sent the invoice yet).
 await Invoice.create({
   payment_request: invoice.payment_request,
   value: value,
   memo: memo,
   settled: false,
   send: false,
   user_id: user_id,
 });

 // Finally, the function returns the invoice that was created with the Lightning service.
 return invoice;
};
```

Update invoiceEventStream in lnd.js

```
const invoiceEventStream = async () => {
 await lnd.services.Lightning.subscribeInvoices({
   add_index: 0,
   settle_index: 0,
 })
   .on("data", async (data) => {
     if (data.settled) {
       // Check if the invoice exists in the database
       const existingInvoice = False;

       // If the invoice exists, update it in the database
       if (existingInvoice) {
         // update db
       } else {
         console.log("Invoice not found in the database");
       }
     }
   })
   .on("error", (err) => {
     console.log(err);
   });
};
```

```
const invoiceEventStream = async () => {
 await lnd.services.Lightning.subscribeInvoices({
   add_index: 0,
   settle_index: 0,
 })
   .on("data", async (data) => {
     if (data.settled) {
       // Check if the invoice exists in the database
       const existingInvoice = await Invoice.findOne(data.payment_request);

       // If the invoice exists, update it in the database
       if (existingInvoice) {
         await Invoice.update(data.payment_request, {
           settled: data.settled,
           settle_date: data.settle_date,
         });
       } else {
         console.log("Invoice not found in the database");
       }
     }
   })
   .on("error", (err) => {
     console.log(err);
   });
```

Update pay invoice endpoint

```
router.post("/pay", authenticateAdmin, async (req, res) => {
 // Extract the 'payment_request' property from the request body.
 const { payment_request } = req.body;

 // Use the 'payInvoice' function to attempt to pay the invoice. This function is asynchronous, so we use 'await' to pause execution until it completes.
 const pay = await payInvoice({ payment_request });

 // If there was an error making the payment, we send back a response with a status of 500 (Internal Server Error) and the error message.
 if (pay.payment_error) {
   res.status(500).json(pay.payment_error);
 }

 // If the payment was successful (indicated by the existence of 'pay.payment_route'), we create a new 'payment' record in the database.
 if (pay?.payment_route) {
   const payment = await Invoice.create({
     // The payment details include the original payment request, a flag indicating it was sent, the total amount, any fees, and the settlement details.
     payment_request: payment_request,
     send: true,
     value: pay.payment_route.total_amt,
     fees: pay.payment_route.total_fees,
     settled: true,
     settle_date: Date.now(),
   });

   // After creating the new payment record, we send a response with a status of 200 (OK) and the details of the new payment record.
   res.status(200).json(payment);
 }
});
```

Test the updated /invoices endpoints

With Postman / Polar

Creating an invoice through the API

- Startup polar just like we did in lesson 6 by opening Docker Desktop then Polar
- Click on your previously created network and start it up (you should still automatically connect to the Alice node configured in your .env)
- Login as {“username”: “Alice”, “password”: “pass1”} and grab the JWT
- Add the JWT to the authorization header for a POST /invoices/create call and we should be able to create a real invoice with our Alice node

POST http://localhost:5501/lightning/invoice | Send | 200 OK | 85.7 ms | 619 B | Just Now

JSON | Auth | Query | Headers 3 | Docs


```
1  {
2    "value": 581,
3    "memo": "test invoice",
4    "user_id": 1
5  }
6
7
8
9
```


Preview | Headers 23 | Cookies | Timeline


```
36        242
37      ]
38    },
39    "payment_request":
"lnbcrt5810n1pjttg6spp5xqnynf35up29087y6595vesulhektvqq9ecqkadm7srafmtwymeq
dq5w3jhxapqd9h8vmmfvdjscqzpgxqyz5vqsp50k4kgns6cv4aspwp85aljj27n0r397mhj4nuy
amav2yq2xfdrrcs9qyyssqvsusrxfmf57tva6gm8rn2xdt0lpc79d649ve5eg5z5sn6p85hs3rx
yh03f5g63n8kmxva0exuqf98p6spenhqq4gmcu0lk8hz86umdgqje5dqg",
40    "add_index": 9,
41    "payment_addr": {
42      "type": "Buffer",
43      "data": [
44        125,
```

Call the GET /invoices endpoint

You should see your newly created unpaid invoice (don't forget the JWT!)

No Environment | Cookies

Filter

POST pleb-node-template

GET get booger

course-2

POST pay invoice

POST create invoice

GET get invoices

GET get balance

GET get user by username

DEL delete user

PUT update user

POST login user

POST register user

GET get users

GET get server

GET http://localhost:5501/lightning/invoices | Send

200 OK | 27.1 ms | 789 B | Just Now

Body | Auth | Query | Headers 1 | Docs

Enter a URL and send to get a response

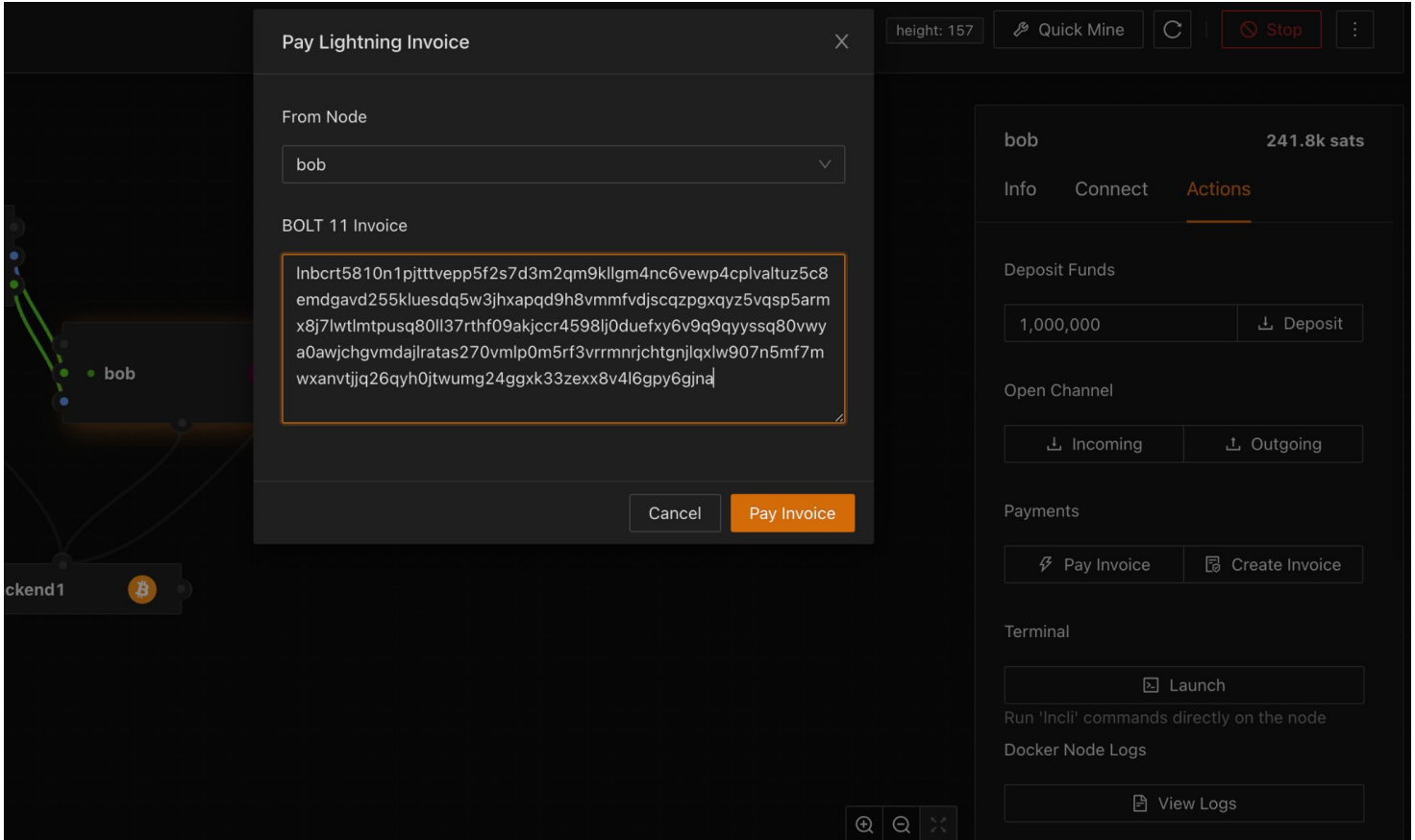
Select a body type from above to send data in the body of a request

Introduction to Insomnia

Preview | Headers 23 | Cookies | Timeline


```
2  {
3    "id": 1,
4    "payment_request": "lnbcrt1u1p...",
5    "value": 100,
6    "memo": "yo",
7    "fees": null,
8    "send": 0,
9    "settled": 0,
10   "settle_date": null,
11   "created_at": "2023-07-16 22:16:47",
12   "user_id": 1
13 },
14 {
15   "id": 2,
16   "payment_request": "lnbcrt2u1p...",
17   "value": 100,
18   "memo": null,
19   "fees": 10,
20   "send": 1,
21   "settled": 1,
22   "settle_date": "2023-07-16 22:16:47",
23   "created_at": "2023-07-16 22:16:47",
24   "user_id": 2
25 },
26 {
27   "id": 3,
28   "payment_request":
     "lnbcrt5810n1pjtttvepp5f2s7d3m2qm9kllgm4nc6vewp4cplvaltuz5c8emdgavd255kluesdq5w3jhxapqd9h8vmmfvdjscqzpgxqyz5vqsp5armx8j7lwtlmtpusq80ll37rthf09akjccr4598lj0duefxy6v9q9qyyssq80vwya0awjchgvmdajlratas270vmlp0m5rf3vrrmnrjchtgnjlqxlw907n5mf7mwxanvtjjq26qyh0jtwumg24ggxk33zexx8v4l6gpy6gjna",
29   "value": 581,
30   "memo": "test invoice",
31   "fees": null,
32   "send": 0,
33   "settled": 0,
34   "settle_date": 0,
35   "created_at": "2023-07-17 21:07:37",
36   "user_id": 1
37 }
38 ]
```

Now go into polar and have the Bob node pay the created invoice

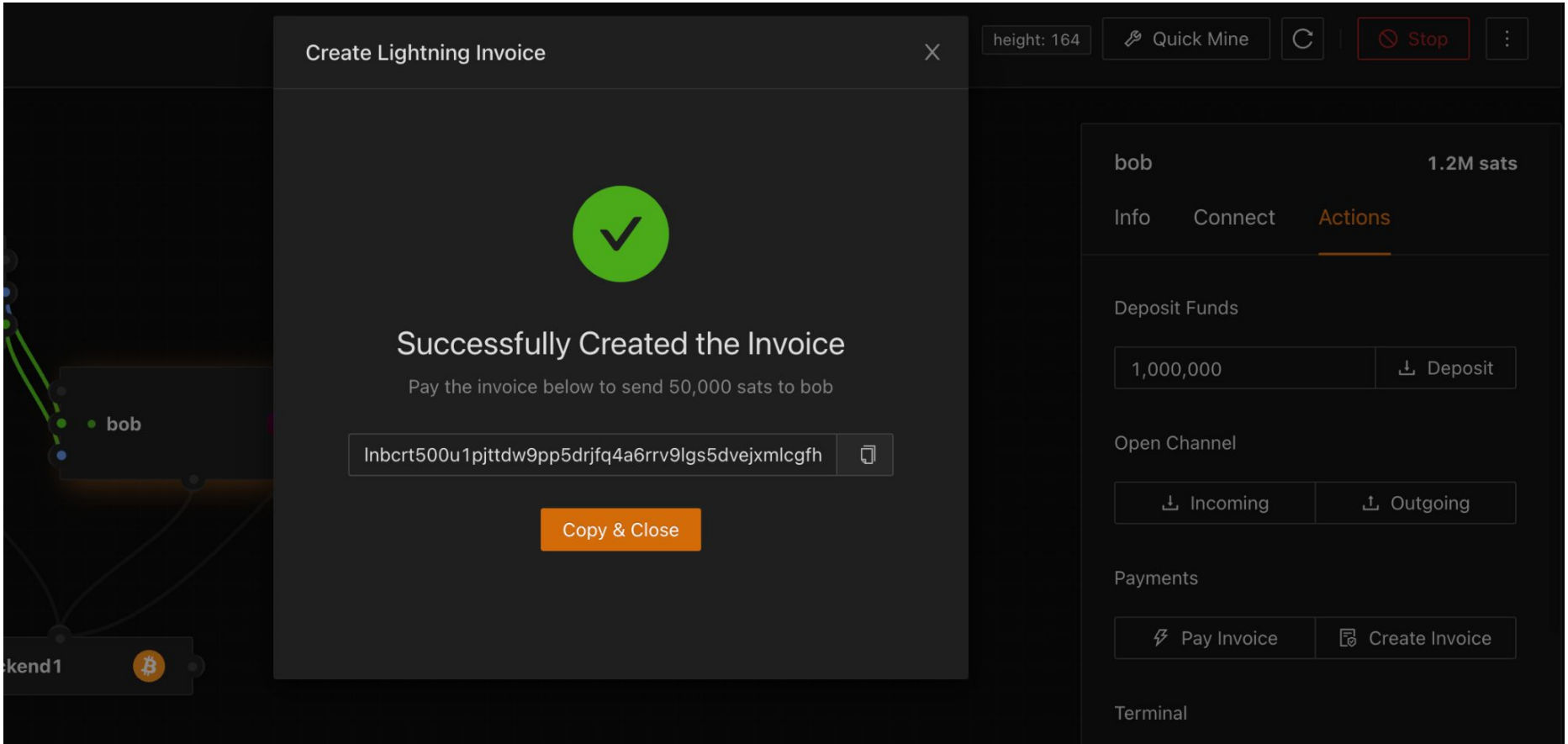


Call get all invoices endpoint again

You should now see that the invoice we created has been paid via the “settled” and “settle_date” properties


```
26  {
27    "id": 3,
28    "payment_request":
"lnbcrt5810n1pjttvcfpp57urtwe6vwmw4ugpqv5vvlgv4s5ex0ewhqat5aya5gac3q6w8073q
dq5w3jhxapqd9h8vmmfvdjscqzpgxqyz5vqsp5j0j84lelkmw4xljxgt8yjnqaq4a08yjrz7qrc
078rgu6vmeqwrms9qyyssq4f6v6exf32rz3e0hhx2xvvn79rg06eyye4e436c5a5ga064pqenxv
plnayawyx2j6xnah7spd2g88txrpdxn0ph4g5599m29x2ypq7sq4tyu43",
29    "value": 581,
30    "memo": "test invoice",
31    "fees": null,
32    "send": 0,
33    "settled": 1,
34    "settle_date": 1689629465,
35    "created_at": "2023-07-17 21:30:49",
36    "user_id": 1
37  }
38 ]
```

Now lets have Bob create an invoice so we can pay it



Send a request to pay an invoice

Add the invoice as the payment request and the user_id of the user paying (Alice)

POST http://localhost:5501/lightning/pay Send 200 OK 349 ms 421 B Just Now

JSON Auth Query Headers 2 Docs


```
{
  "payment_request":
"lnbcrt5810n1pjttjnypp5me2kg9nhewpka0f5kh46yuxulf6685wjh34ch3e69yxj2d04ajpsdq
qcqzpgxqyz5vqsp5w7y4u7cz2aqatr99rsy87p6tsx2uyzcfux4q5r5s6gwlq6ck0n3q9qyyssq74
5cdpwt6klc6t78qmjr9vkulqt3wrex3s9a0ags9f5zq6rm9emx5f3evpl35v90j23565mhtpx68kn
qs7kseaawh0fjv979m35pc5qpmu4nat",
  "user_id": 1
}
```


Preview Headers 23 Cookies Timeline


```
[
  {
    "id": 4,
    "payment_request":
"lnbcrt5810n1pjttjnypp5me2kg9nhewpka0f5kh46yuxulf6685wjh34ch3e69yxj2d04ajpsdq
qcqzpgxqyz5vqsp5w7y4u7cz2aqatr99rsy87p6tsx2uyzcfux4q5r5s6gwlq6ck0n3q9qyyssq74
5cdpwt6klc6t78qmjr9vkulqt3wrex3s9a0ags9f5zq6rm9emx5f3evpl35v90j23565mhtpx68kn
qs7kseaawh0fjv979m35pc5qpmu4nat",
    "value": 581,
    "memo": null,
    "fees": 0,
    "send": 1,
    "settled": 1,
    "settle_date": 1689635443505,
    "created_at": "2023-07-17 23:10:43",
    "user_id": 1
  }
]
```

We have now fully connected the API and Database

And tested it!