PASSWORD-LESS AUTH

Open Source SMS/Email based authorization via tokens and secure Azure SQL.

HOW IT WORKS

Customize the code for your own Azure SQL database, Twilio account, and Mailgun account.

Customize a few sql statements in the UserSqlContext.cs file to match the way your database is setup.

Completely forget annoying passwords and the security aspect of storing them.

THE FLOW

Register

- POST request with our User Model via raw JSON to our register validation endpoint:

  /api/user/register/validate

  -> RETURNED true / false if code sent

- POST request with the Token your user enters in your application to our registration auth endpoint:

  /api/user/register/auth

  -> RETURNED user we POSTed with an Active Registration Session

Login

- POST request with our User Model via raw JSON to our login validation endpoint:

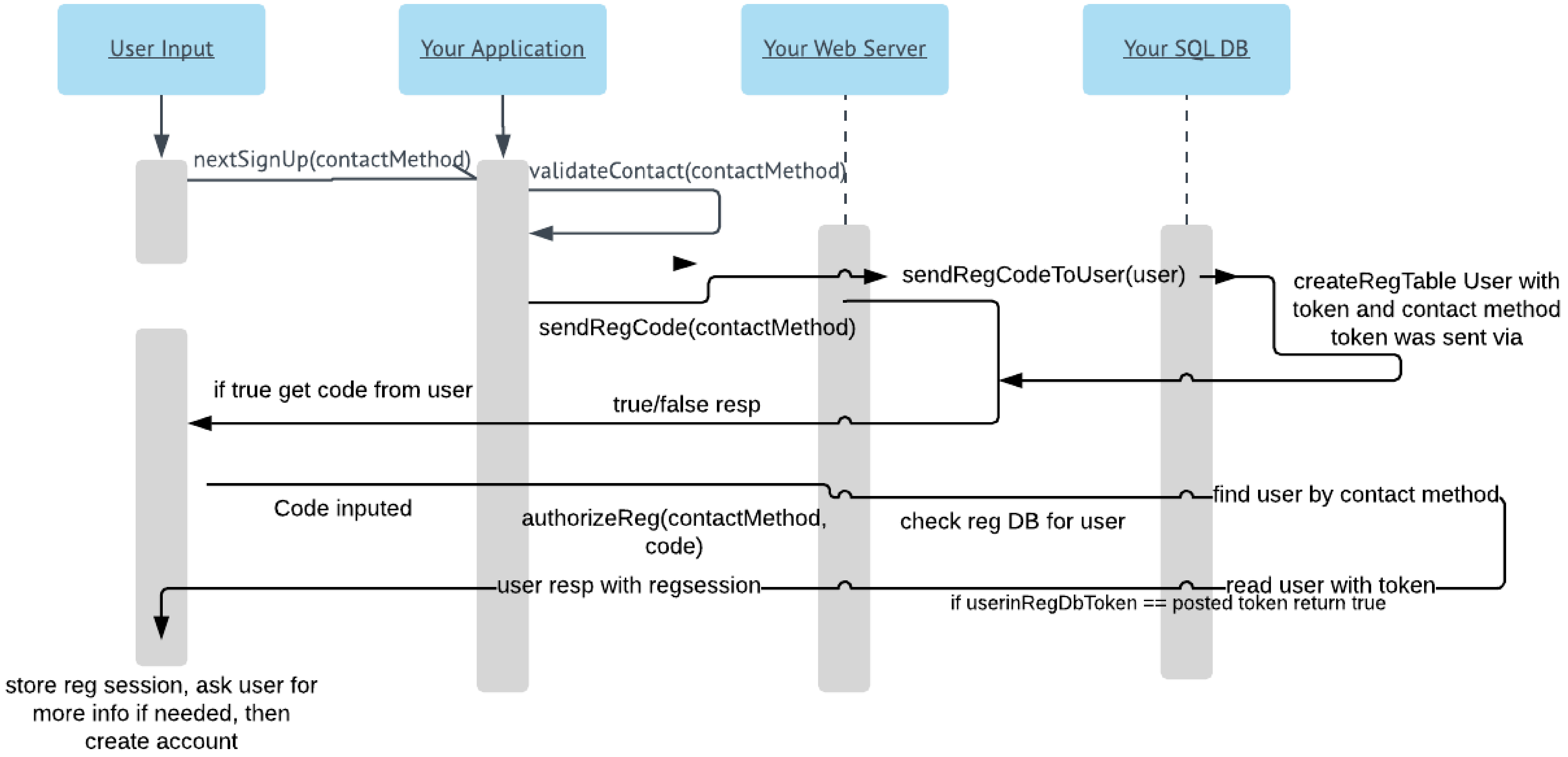
  /api/user/login/validate

  -> RETURNED true / false if code sent

- POST request with the Token your user enters in your application to our login auth endpoint:

  /api/user/login/auth

  -> RETURNED active user in Azure SQL DB with active login session


User Input
Your Application
Your Web Server
Your SQL DB
nextSignUp(contactMethod)
validateContact(contactMethod)
sendRegCodeToUser(user)
createRegTable User with token and contact method token was sent via
sendRegCode(contactMethod)
if true get code from user
true/false resp
Code inputed
authorizeReg(contactMethod, code)
check reg DB for user
find user by contact method
user resp with regsession
read user with token
if userinRegDbToken == posted token return true
store reg session, ask user for more info if needed, then create account

SETTING IT UP

You will need a twilio account, a mailgun account, and an Azure account with an active payment plan (either free or paid).

CREATING SQL TABLES

- Create a new sql database in azure
- Go into query editor
- *note* testReg, testDebug, and testUsers are all just table names, replace these in the table commands with your own names.
- Create registration table (order is especially important in this table):

```
CREATE TABLE testReg (
USERID int NOT NULL IDENTITY(1,1) PRIMARY KEY,
REGISTRATIONID varchar(255) NULL,
TOKEN varchar(255) NULL,
EMAIL varchar(255) NULL,
PHONENUMBER varchar(255) NULL,
);
```

- Create Debug table (order is especially important in this table):

```
CREATE TABLE testDebug (
DEBUGID INT NOT NULL,
CONSOLEWRITE VARCHAR(255) NOT NULL,
);
```

- Create User table (if you change structure refer to Altering UserSqlContext.cs slide):

```
CREATE TABLE testUsers (
USERID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
LOGINSESSION VARCHAR(255) NULL,
FIRSTNAME VARCHAR(255) NOT NULL,
LASTNAME VARCHAR(255) NOT NULL,
EMAIL VARCHAR(255) NULL,
PHONENUMBER VARCHAR(255) NULL,
);
```

ALTERING APPLICATIONSETTINGS.CS

- ApplicationSettings.cs is where all of your login info resides. I recommend adding */ApplicationSettings* to your .gitignore file if you plan on uploading this to git, as it will contain sensitive information.
- Alter the strings in the constructor to match your own login information and save the file.
- When adding the ConnectionString in GetConnectionString() make sure to remove the '{}' in the User ID and Password field. Azure will provide you a string and in it will be something similar to: 'User ID={yourId};Password={yourPassword};'
- MAKE SURE TO REMOVE THE CURLY BRACES SO IT would look like this:
- User ID=mySqlId;Password=mySqlPass;
- **NOTE this is not the full connection string, just the only part that needs to be altered after copying and pasting from your azure portal.

ALTERING USERSQLCONTEXT.CS

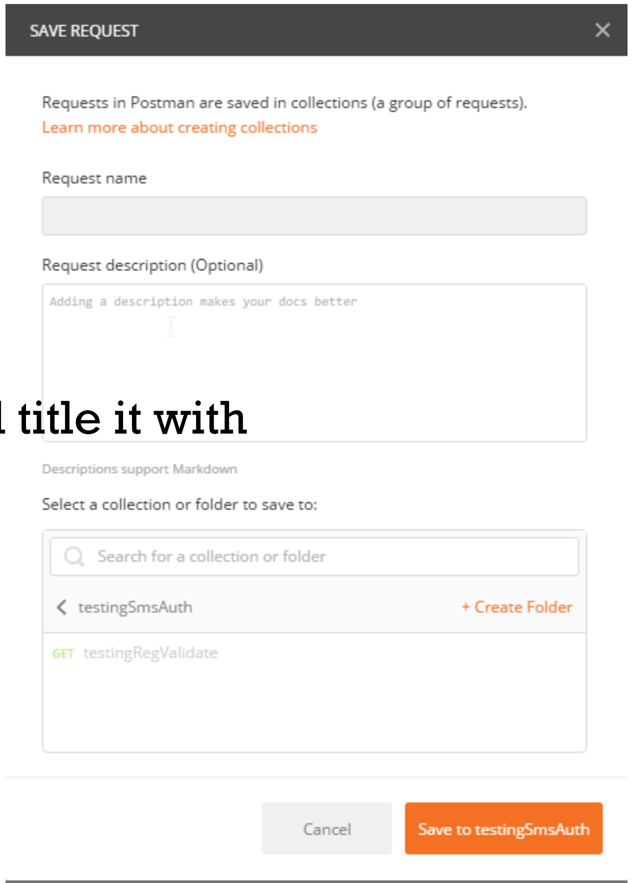
- Depending on the way you want to setup your SQL data structure, you will need to alter a few functions in the UserSqlContext.cs file.
- First things first set the public const strings with your various Debug, User, and registration table names.
- Next set the private const strings to the names of your SQL columns in your various tables.
  - E.g. PhoneSqlKey should be set to the NAME of your Phone Number varchar column in your User table.
- Finally alter various statements to fit with your sql data structure you create. You will need to alter the GetUserFromReader(), GetUserFromReaderAsync() functions at a minimal to match the structure of your database. Match the order and types with the order and types of the columns you used in your create statement. For more help with this step watch the video on Youtube. (link coming soon)

TESTING IT ALL

In this section you'll discover how to test your new SMS/email based authorization REST API

TEST WITH POSTMAN

- Postman is an application available for free on Windows, Mac, and Linux that allows you to send different types of HTTP requests to a custom server endpoint.
- Download postman on their website: https://www.getpostman.com/
- Install postman on your operating system
- Open Postman, Select New (top left), Choose Request.
  - Give the request a name, e.g. testingRegValidate
  - Optionally add a description
  - Either select an existing collection or create a new collection of requests and title it with your app name, e.g testingSmsAuth
  - Tap 'Save to testingSmsAuth' or whatever you named your folder.
  - -> proceed to the next slide for more info


SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).
Learn more about creating collections

Request name

Request description (Optional)

Adding a description makes your docs better

Descriptions support Markdown

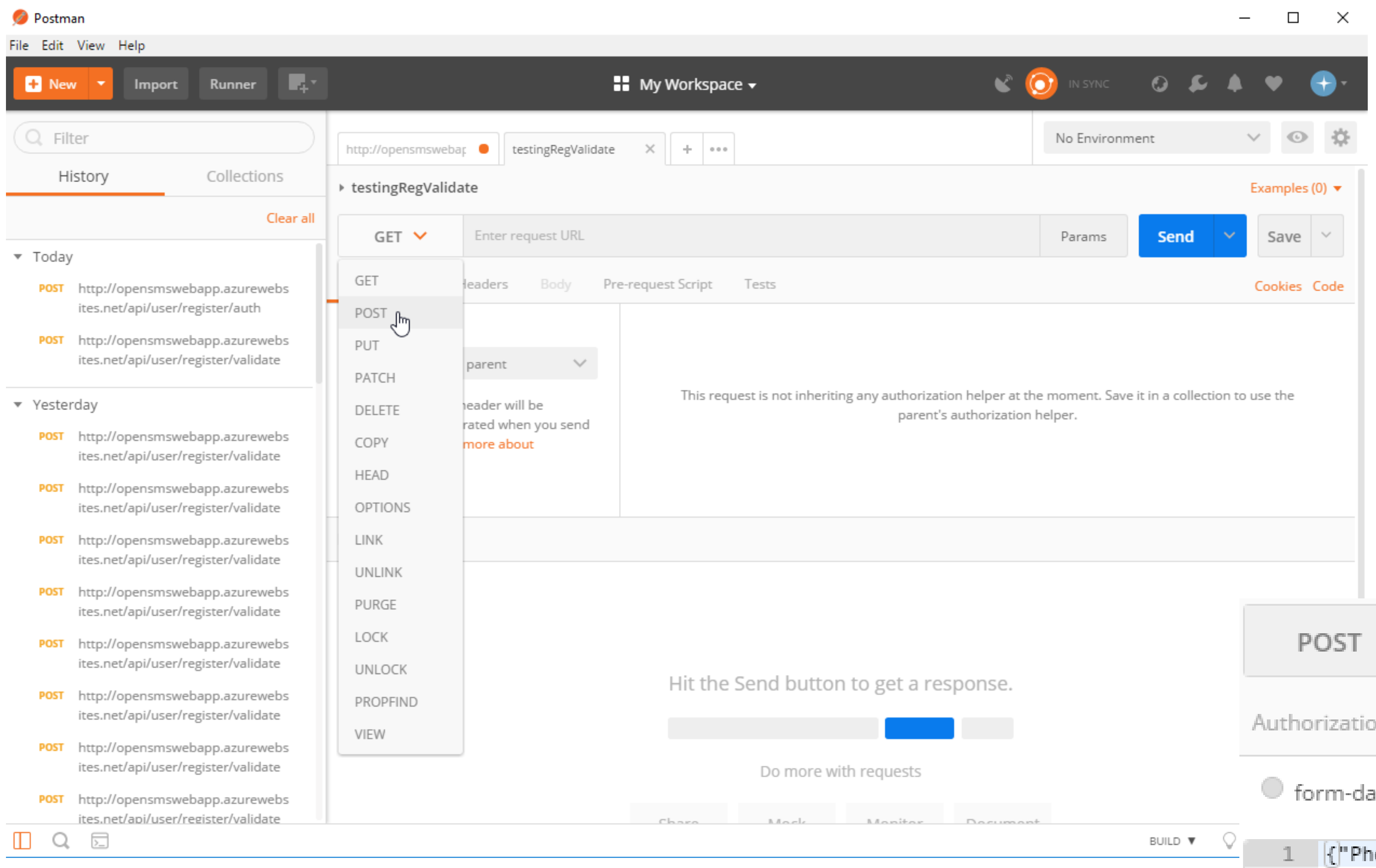
Select a collection or folder to save to:

Search for a collection or folder

testingSmsAuth + Create Folder

GET testingRegValidate

POSTMAN CONTINUED



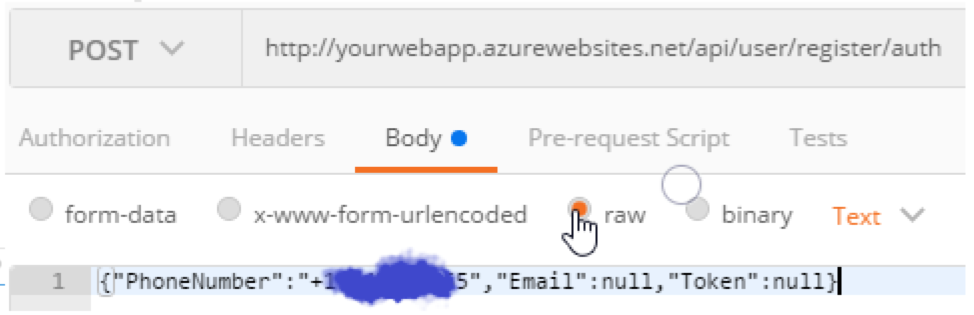
POST http://opensmswebapp.azurewebsites.net/api/user/register/auth

POST http://opensmswebapp.azurewebsites.net/api/user/register/validate

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Hit the Send button to get a response.

- Select the type of request from the middle left, select POST.
- Click on 'Body' (no longer greyed out after selecting 'POST') directly under where you enter the request URL
- Click on 'raw' under body
- Enter a basic JSON string to test your end points



POST http://yourwebapp.azurewebsites.net/api/user/register/auth

Body: raw, Text

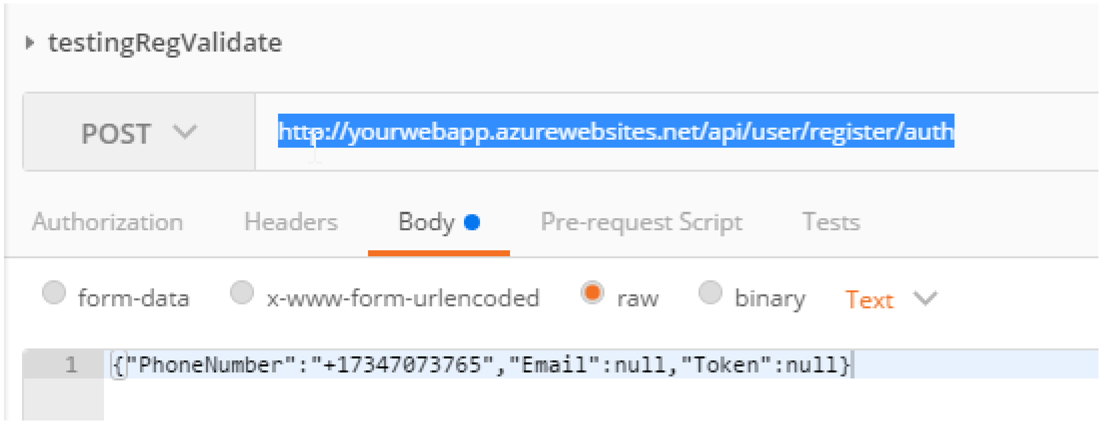
```
{"PhoneNumber":"+1[redacted]5","Email":null,"Token":null}
```

POSTMAN CONTINUED…

- Input a basic JSON string with your phone number to test it, email and token are optional, I would just set them to null:

{"PhoneNumber":"+17347073765","Email":null,"Token":null}

- Alter the endpoint postman is pointing towards to your azure web app endpoint (or your local server) please note if you choose to debug on a local server you will need to configure your azure SQL firewall to allow your IP address to access the database, otherwise you will get authentication errors when debugging locally. I recommend creating an azure web app and publishing there for debugging, minimal configuration is required when this is done. Follow the video tutorial for more information.


testingRegValidate

POST http://yourwebapp.azurewebsites.net/api/user/register/auth

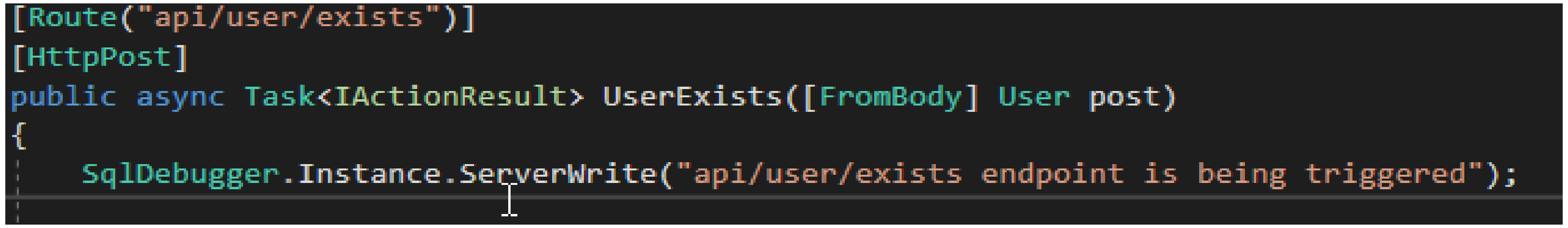
Body: raw, Text

```
{"PhoneNumber":"+17347073765","Email":null,"Token":null}
```

- Finally press Send, Postman will take a second and then should return something in the response. It will either return a JSON response (your connection to your web app is working) or an error. View the next page for typical error messages.

TYPICAL POSTMAN ERROR MESSAGES

- 404 not found error – This means postman can't find your endpoint on the internet. Try entering your endpoint in a browser and see if you get the same error, if you do check your spelling / to make sure your endpoint published correctly.
- 500 internal server error – This means your server failed to return a response, this could be do to an unhandled exception being thrown. Most likely this means there is an error with your SQL code. Check that the controller is even being reached by adding a debug statement to the first line of your controller. Republish the webapp and send the request again. You should see your debug message appear in your SqlDebugTable, you will still be returned the 500 error, but this will give you a base point for debugging. You can then explore the functions within that controller with similar debug statements, through trial and error you should be able to debug the failing endpoint. View the video tutorial for more in depth 500 error debugging.



```
[Route("api/user/exists")]
[HttpPost]
public async Task<IActionResult> UserExists([FromBody] User post)
{
    SqlDebugger.Instance.ServerWrite("api/user/exists endpoint is being triggered");
```

NO ERROR MESSAGES? JSON RESPONSE?

- If you received a JSON response congratulations, that means the endpoint is working. If you received a text/email at the contact method you specified, a celebration is in order. If you did not receive a text/email, some further debugging is required.
- If you did not receive the code, there is likely an issue with your Twilio (sms) or Mailgun (email). Check your SMS logs on Twilio and see if you received any requests to send out messages. Same with Mailgun if you provided an email instead of a phone number.
- If you didn't receive any requests to send a message in either log, there is an error with your ApplicationSettings.cs configuration of Mailgun or Twilio. Check your SmsDebug log to see if there's any more information. If an exception was caught it will be automatically written into your debugLog assuming you set it up your debugTable correctly. View the video for more help.

GOT THE CODE?

- Awesome! Now test the authentication endpoint. Replace /validate with /auth in the endpoint Postman is targeting. Change the raw JSON statement we are posting to the endpoint to:

  {"PhoneNumber":"+17347073765","Email":null,"Token":"tokenWeGotSent"}

  Replace 'tokenWeGotSent' with whatever token you actually were sent

  e.g. '4324' : {"PhoneNumber":"+17347073765","Email":null,"Token":"4324"}

  Press send again, you will either receive an error (refer to postman error messages slide again) or you will be returned the registration session. If the registration session is null, there is an error with your code. Refer to the No Error Messages slide.

REGISTRATION SESSION NOT NULL?

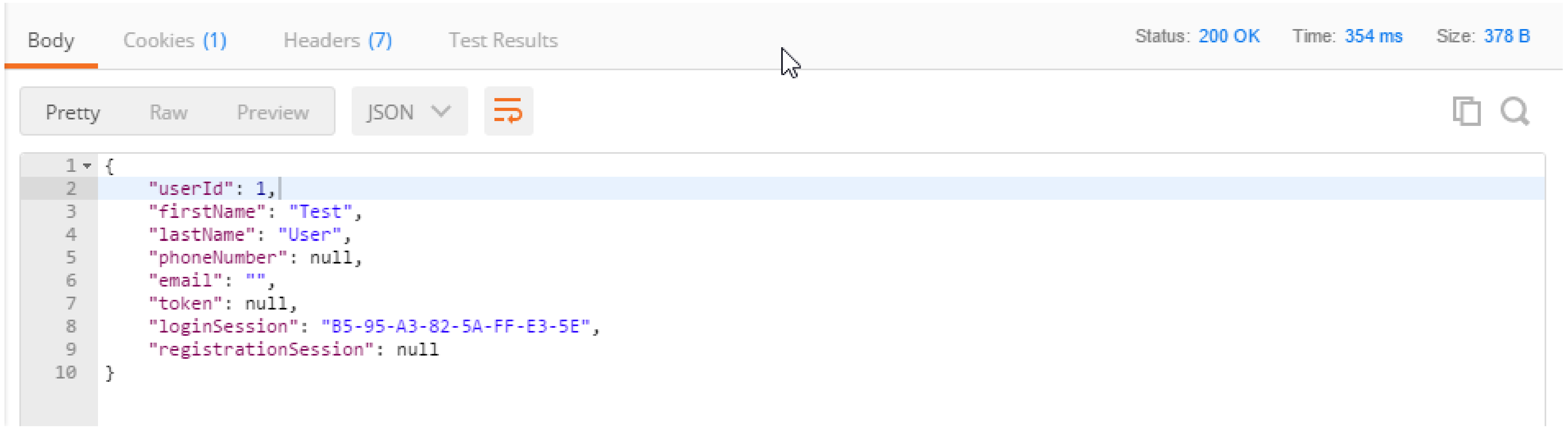
- Great! The registration session endpoints are working! Now test the user creation endpoint. Please refer to the same debugging cycle we adopted for the registration endpoints for all endpoints we will test from now on.
- To test the creation endpoint, we will need to post a full User JSON object in the same manner we have been posting them, we will just need to add more information to our raw statement.
- If you used the same model for the user as we did, this JSON statement will work for you. Otherwise you will have to customize it to your custom data structure / model. Refer to the video for more information.

```
{"CompanyName":"Cutflo","CompanyMailingUrl":"cutflo.io","ApiKey":"TESTKEY1","FirstName":"Test","LastName":"User","PhoneNumber":"+17347073765","Email":null,"RegistrationSession":"theRegistrationSessionYouJustReceived"}
```

- Note the order does not matter in the JSON statement, just make sure to include all column values in your SQL table that are NOT NULL, the way we set OUR table up contact methods can be null, thus if the user only wants to provide one method they can. Notice '"Email":null' in our statement, since we authenticated via phone number. C# will default any missing parameters to null so statements like these aren't important, it just highlights the difference. What's most important is including the parameters that are supposed to be NOT NULL in your table.
- Make Sure to replace the RegistrationSession with the registration you just received when testing your authentication end point.

USER CREATED

- If you received a JSON response with your User data, Great!
- If you did not, refer to our previous debugging steps.
- You should receive a response similar to this:



```
{
    "userId": 1,
    "firstName": "Test",
    "lastName": "User",
    "phoneNumber": null,
    "email": "",
    "token": null,
    "loginSession": "B5-95-A3-82-5A-FF-E3-5E",
    "registrationSession": null
}
```

NOW TEST THE LOGIN ENDPOINTS

- Now that you have successfully created and tested your registration endpoints, it’s time to test your login endpoints.
- The way they work is very similar, but instead it pulls from our user table instead of our registration table. Once you have created your user in your user table. Test the login validation endpoint:
  - /api/user/login/validate
  - The response should be ‘success : true/false’ with the true or false indicating if the token was sent successfully
- Test the authentication endpoint once you get a true response and receive a code. If you have any issues refer to our previous debugging cycle.
  - /api/user/login/auth
  - The response should be similar to when you created your account. You should receive the entire user’s information in the same format as your User.cs class, just in JSON.

YOU'RE DONE!

- There's a few more endpoints you can mess around with, but those are the necessary ones for registration/logging in.
- I won't include tutorials for these endpoints (such as /api/user/exists) so that you can mess around with them and learn more about REST API's.
  - (also I'm just lazy)
- Want to hire me as a Xamarin or .NET consultant, part time or full time? Contact me at bennett@cutflo.com