Process Report Phase 3

Popularity and applicability of backend development programming languages.

M.M. Wichrowski - 18136753

Autor

Student number	18136753
Surname	Wichrowski
Initials	M.M.
Name	Mikolaj

Shool

Name study career counselor	Peter Becker
Name expert examinator	Fatih Caglayan

Company

Name	Incubeta
Department	Cloud City
Address	Westeinde 3A
Zip code	2275AA
City	Voorburg
Phone number	070 300 1950
URL	https://dqna.com/

Client

Surname	Farkas
Name	Szabolcs
Position	Head of Development
Email	szabolcs@incubeta.com

Processed feedback

- I have not yet corrected the grammar in my documents. This is due to time constraints, but it will be taken care of in the beginning of the next phase.
- I have expanded the desk research with my motivation for the selection of the three chosen programming languages.
- I have expanded the application design document with a description of how the use was constructed, why and how it was validated.
- I wrote down what I have learned during the development of the prototypes. I also described the feedback I have received on the executed work. This indicates what I have learned and how I validated my learnings. See chapter “Professional duties”.

Delivered products

1. Artefacts: Backend codebases for different programming languages and frameworks.
2. Artefact: Frontend codebase with Integration of backend APIs in the frontend proof of concept

Important notices

- The backend code bases are delivered in a zip file and contain four folders and one file.
- Three folders are the backend prototypes and can be identified by the suffix.
    - sharp = C#
    - js = Javascript/Node.js
    - py = Python
- In the three backend codebase folders there is a “sources.md” file which lists all sources used to create the application.
- In the three backend codebase folders there is a “steps_taken.md” file which lists the steps I have taken per user story to implement the story as code.
- The postman file is the testing file which contains the code to execute the tests. To execute the tests, the backend must be started. There is no documentation yet on how to start the application, this will be delivered on may 19th to the client and in the next phase on onstage.
- The frontend codebase is still under construction and is missing most of the functionalities that are described in the user stories. The suffix for the folder is “frontend”.

Glossary

User story:
A user story is an informal, general explanation of a software function that is written from the perspective of the end user.

Framework:
A framework is a set of software components that can be used when programming applications, but the agreements on how those components are used within a group of developers and which code standards and libraries are used can also be part of a framework.

Dependencies:
A software dependency is an external standalone library that can be as small as a single file or as big as multiple files and folders organised into packages to perform a specific task.

ORM:
Object–relational mapping (ORM, O/RM, and O/R mapping tool) in computer science is a programming technique for converting data between type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

Http:
HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser.

Endpoint:
A web service endpoint is a web address at which users of a specific service can gain access to it. By referencing the URL, users can get to operations provided by that service. In addition a method can be provided in the request to handle specific actions. For example GET to retrieve data, POST to create and DELETE to delete data.

Dependency injection:
Dependency injection is the act of adding an instance of object x to object y without explicitly creating a new object within the scope of object y. Meaning we give the object it needs to run certain operations without making the dependency closely coupled. This is called loose coupling because one object is no longer dependent on the other and can exist without it. If necessary we can pass another object to it instead.

Professional duties

D1: Realisation software

I have started the development of three prototypes using a fictional use case that is similar to our usual work. I have done this by using the kanban methodology. The process entails executing one user story at a time and making the implementation in each programming language. This way I was able to steer the delivery in case I underestimated the amount of work or forgot any tasks. This means I can still deliver a working application and therefore come to a conclusion for the research. Though the application might miss functionalities, the programming languages would still be tested. Fortunately all of the user stories are completed in each programming language, tested and reviewed by experienced engineers.
Besides the backend proof of concepts I have also worked on a frontend application that will indicate the possibility of using the three programming languages as a backend for our web development. The frontend application is still in development but will be completed before the 19th of may. After this the solution will be presented to the head of engineering for technical to give insights on how web applications can be developed in the different programming languages.

Gf: Learning to learn

During the third Phase I was mainly spending my time on the development of three proof of concepts. These are necessary for the next phase which will consist of making a suggestion report and constructing an opinion on the programming language that I think we should use. To achieve this I also need some experience with the programming language and be able to show that it is possible to do web application development in the selected programming language. Since I did not know Node.js and C# indepthly as a programming language, I have contacted experienced developers to aid me in learning these languages and how to use them professionally.

Python
The first programming language I have chosen to compare is Python. This programming language was already known to me since we have used this for work and I have 5 years experience with. This programming language took me the least time to create a proof of concept with. I have initially overlooked the scaffolding that had to be done for the project, but luckily this is easy with Python. The scaffolding that had to be done, was setting up an existing framework and configuring the development environment. We can reproduce this by following the steps below:

- Make a folder and install Python and a tool called pipenv. This is used to encapsulate dependencies in an environment. It allows us to develop without influencing or being influenced by other Python projects.
- Use pipenv to create a virtual environment. This is the so-called encapsulated environment where we store our dependencies.

- Install Django using the documentation https://docs.djangoproject.com/en/4.0/topics/install/. The Django framework is made in Python and is used to make development standardised. Besides this Django contains tools which allow us to communicate with a database and expose interfaces through http. It is not relevant for the comparison of the programming languages to use Django since we are exploring the programming language itself, regardless it is beneficial to sometimes use a framework like this, since it gives us tools and guidelines that allow for more standardised development.
- Create a project with the command `django-admin startproject capsaicin_events` . For more information see the documentation on https://docs.djangoproject.com/en/4.0/intro/tutorial01/
- Create the api folder with the `./manage.py startapp api` command. This folder will contain our source code.
- Install the django rest framework. This will allow us to create a REST api. Follow the instructions on https://www.django-rest-framework.org/#installation to install this dependency.
- Configure installed applications. See the documentaion at https://docs.djangoproject.com/en/4.0/ref/applications/ for instructions.
- Create and execute the migrations to populate a testing database with data. Read the documentation on https://docs.djangoproject.com/en/4.0/topics/migrations/ for instructions.
- Create a super user so that i can log in to the django backend. Follow the documentation on https://docs.djangoproject.com/en/4.0/ref/django-admin/#django-admin-createsuperuser to do so.
- Start the server to test if it works. Use the `./manage.py runserver` command to do so.
- Go to your browser and navigate to http://localhost:8000/admin/. You will see a login screen where you can enter your super user credentials. This will redirect you to the administrator interface.

After completing this setup I was able to start developing features. The first feature. This is the event creation, but to be able to create events we also need users. So I started by making the necessary models that are used for the ORM. The ORM is part of django and is used to get, delete and create data in the database, in a way that is usable by our code. The first two models are the user and event model and they are an almost direct representation of the ERD diagram. The diagram can be found in the Design document artefact.

```
class User(models.Model):
    username = models.CharField(max_length=255, unique=True)
    password = models.CharField(max_length=255)


class Event(models.Model):
    creator = models.ForeignKey(User, on_delete=models.CASCADE)
    description = models.CharField(max_length=255)
    picture = models.CharField(max_length=4000000)
    location = models.CharField(max_length=255)
```

Now the code can interact with the database, but we also need to expose these models to the outside world. We do this by creating views, which make the code accessible through HTTP.

```
class EventViewSet(viewsets.ViewSet):
    queryset = Event.objects.all()
    permission_classes = [Protected]
    renderer_classes = [CamelCaseJSONRenderer]

    def list(self, request):
        serializer = EventSerializer(self.queryset, many=True)
        return Response(serializer.data)

    def create(self, request):
        data = {
            **request.data,
            "creator": request.COOKIES["user_id"]
        }
        serializer = EventCreateSerializer(data=data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        data = serializer.data
        event = get_object_or_404(self.queryset, pk=data.get("id"))
        response_serializer = EventSerializer(event)
        return Response(response_serializer.data, status=201)
```

Within the code we case a class and methods indicated with the *def* keyword. This indicates it is a function that can be executed through HTTP by using the correct method. If we want to list all of the events we use the HTTP GET method. Then the function will run, clean the data according to a serializer definition and return it as a response. If we want to create an event in our database, we use the create function. To execute it we need to use the HTTP POST method and supply data. This will execute a function that gets this data, validates it using a serializer and then save it in the database. After this it will return the newly created entry and return the newly created data as a response. In order to make this work we need the correct serializers.

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username']


class EventSerializer(serializers.ModelSerializer):
    creator = UserSerializer()

    class Meta:
        model = Event
        fields = ['id', 'creator', 'description', 'picture', 'location']
```

The serializers are used to define what model is used and what fields we want to receive and return as a response. In the case of an event we need the general information like description and location, but also the related user. This is defined as seen above. This serializer will automatically define what the response or request must look like based on the data supplied.

All other entities have been created in a similar way and can be viewed in the backend codebase artefact under the Python folder. To ensure this code was constructed correctly I have asked feedback from an experienced python developer. He has given me feedback, mainly on naming of certain functions and classes. He also has given me feedback on the Django features I have used. One example was my usage of a so-called HyperlinkedModelSerializer instead of a ModelSerializer. The difference being that my initial implementation is used for responses that contain a lot of data and must therefore be chunked into pages. The ModelSerializer, that the Python engineer advised, is used when making HTTP endpoints that do not need this functionality. I did not initially know this and have learned this way that simplicity is important when programming in Python. Besides this I also learned how to make the Python code more simple and readable.

Node.js

For the second language to compare I have chosen Node.js. Node.js is javascript for the backend. I have used javascript for frontend but not a lot in backend development. To make sure I have a good start, I have contacted one of my colleagues who is an experienced Node.js developer and has more than five years experience in Node.js development. He has guided me in setting up a Node.js backend application. He has shown me what his formal companies used when developing backend services in Node.js. The companies he referred to used Nest.js as a framework to make sure there is standardisation within the team. We can set up a project by doing the following:

- Install Node.js from https://nodejs.org/en/ and a tool called npm like instructed on https://www.npmjs.com/package/npm
- Initialise a nest js project, install all packages and typescript by following the instructions on this article https://www.thisdot.co/blog/introduction-to-restful-apis-with-nestjs. Typescript is used to enhance our development experience by making javascript more predictable with type annotations. This allows us to see what type of data is used where. This is needed because javascript does not have type definitions.
- Install and initialise an ORM as described here https://www.prisma.io/docs/getting-started/setup-prisma/add-to-existing-project/relational-databases-typescript-postgres. We use Prisma as an ORM because it is compatible with Node.js and it generates the code we need to interact with the database. We can use any available ORM, but I have used this one since it was advised by the experienced Node.js engineer.

Unlike Python Django, we cannot test if the setup is working without making any code. The first thing we need are the models we use to interact with the database. We do this with prisma definition files. The code needed for the first user story is the following:

```
model User {
  id       Int        @id @default(autoincrement())
  username String     @unique
  password String
  events   Event[]
  attender Attendee[]
  reactions Reaction[]
}

model Event {
  id          Int        @id @default(autoincrement())
  description String
  picture     String
  location    String
  creatorId   Int
  creator     User       @relation(fields: [creatorId], references: [id])
  attendees   Attendee[]
  File        File[]
  Reaction    Reaction[]
}
```

We can see the code looks similar to the Python implementation, yet it has some syntactical differences. For example the brackets. This is only true for these model files, since the rest of the code will look different as you will see. If we want to interact with the database from code we need to create a service. This service is a definition of what we can do with the database in the rest of the code.

```
import { Injectable } from '@nestjs/common';
import { PrismaClient } from '@prisma/client'
import { EventWithCreator } from '../../prisma/types';

const prisma = new PrismaClient()

@Injectable()
export class EventService {
  async createEvent(
    description: string,
    picture: string,
    location: string,
    creatorId: number
  ): Promise<EventWithCreator> {
    return await prisma.event.create({
      data: {
        description,
        picture,
        location,
        creator: { connect: { id: creatorId } }
      },
      include: {
        creator: true,
      },
    })
  }

  getEvents(): Promise<EventWithCreator[]> {
    return prisma.event.findMany({
      include: {
        creator: true,
      },
    })
  }
}
```

The code interacting with the database is a class that has different methods. Each method can do one thing, like for example create an event or get an existing event. In Nest.js we use the “Injectable” keyword above this class to show that we can use it for dependency injection. Besides this service we also need type definitions which tell us what we can expect to get back from the method. This is true for all methods used in further code. And can be seen after each method name. In this case it is “EventWithCreator”.

To expose the service to the outside world we will need a controller. A controller is a class that defines all the HTTP endpoints and business logic.

```
@Controller("event")
export class EventController {
  constructor(
    private readonly eventService: EventService
  ) { }

  @Get("")
  async list(): Promise<EventWithCreator[]> {
    const allEventObjects = await this.eventService.getEvents()

    return allEventObjects.map(({ id, creator, description, picture, location }) => ({
      id,
      description,
      picture,
      location,
      creator: {
        id: creator.id,
        username: creator.username
      }
    }));
  }
}
```

We define the controller by adding the “Controller” keyword on the top of the class. This way the framework will know that it is a controller and all the functions must be exposed as HTTP endpoints. The functions work the same as in Python, though they have a syntactical difference. In these functions I have used an asynchronous approach. This means that each line of code is executed at the same time, unless I explicitly tell it to wait with the “await” keyword. Inside the functions we can call our services which are injected into the class through the constructor as we can see on the fourth line. Then in our functions we take this service and execute the function to interact with our database. This service injection is made possible by the module.

```
@Module({
  imports: [],
  controllers: [EventController],
  providers: [UserService, EventService],
})
export class EventModule {
  configure(consumer: MiddlewareConsumer) { }
}
```

The code to bundle the services and controllers together is called a module. It injects our service into the controller and allows us to organise our application in interchangeable modules. This means we should be able to remove or add new modules without breaking the current code. We indicate a module with the “Module” keyword. This tells Nest.js that it is a module and must be used in the server. After making the code we can start the server by running the npm run serve command. Then we can access the endpoints on http://localhost:3000.

Though I had experience with javascript for frontend development, I still had a lot to learn with backend development in Node.js. Syntactically the code looked the same so I could easily find my way in understanding the syntax.

Despite this, it was still difficult to understand the async and await principles since I have mainly been using a technique called Promises. In the end I have learned that both the async and promise based methods are technically the same. The difference is how the code is executed. Instead of the code being run sequentially like in the promise based method, it is executed at the same time. This means we have to declare where we want to wait for other code.

I also received a lot of feedback and this helped me to learn how to use this programming language and framework more correctly. Javascript is a multi-paradigm programming language. This means that we can do a functional, procedural and object oriented approach. This freedom in the language made it difficult for me to understand what should be used. Some feedback I have received is to use a more object oriented approach and keep in mind how the framework tries to restrict the development to this approach. A lot of my code was functional and not how the framework should work. One example is that I have used functional methods and mapped them directly as a route, instead of using the controller, service and module. After changing this I also made the mistake of not organising my folders correctly. The Nest.js framework advises to have a root folder and within this folder a specific endpoint folder. This could for example be an event or user folder. Within this specific folder we define a service, controller and module. After this we can use the modularity of the framework to attach it to other modules through dependency injection. This means that the framework focuses on using an object oriented approach. All of this is important because a common structured way of working could help my fellow developers understand my code better.

With some guidance and feedback from the experienced developer I have constructed the whole fictional use case as a backend service. Meaning I have completed the Node.js proof of concept, learned how to use it and ensured that my learnings are of some quality.

C#

As the third language I have chosen C# and this was by far the most challenging of the three. Despite having a experienced C# developer at my disposal, the installation of the C# The environment took the longest. The creator of C# is Microsoft and they have a monopoly in terms of available frameworks and tools to create C# web applications. To make a web application that can be hosted on any server, they advise dotnet core. Because the installation was so complicated and long I will not describe how to reproduce this. I followed the documentation on https://docs.microsoft.com/en-us/dotnet/core/install/linux. After having some basic understanding on how to create a C# dotnet core project, I could start developing the features. I started by making the database classes for the first user story. Similar to what I did in Python and Node.js.

```
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

namespace capsaicin_events_sharp.Entities;


public class Event
{

    [Key]
    public int id { get; set; }

    [Required]
    public User creator { get; set; }

    [MaxLength(255)]
    public string description { get; set; }

    [MaxLength(4000)]
    public string picture { get; set; }

    [MaxLength(255)]
    public string location { get; set; }
}
```

The classes look a lot like Node.js classes except for the special decorators we put above each property. These indicate what rules it needs to abide by in the database. We call this a constraint. Then to expose this code to the outside world we have controllers.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Net.Http.Headers;
using capsaicin_events_sharp.Entities;

namespace capsaicin_events_sharp.Controllers;

[ApiController]
[Route("/api/[controller]")]
public class EventController : Controller
{
    private readonly ILogger<EventController> _logger;

    public EventController(ILogger<EventController> logger)
    {
        _logger = logger;
    }

    [HttpGet]
    public IEnumerable<EventResponseType> List()
    {
        IEnumerable<EventResponseType> events;
        using (var context = new AppContext())
        {
            events = context.Events
                .Include(row => row.creator)
                .ToList()
                .ConvertAll(row => new EventResponseType{
                    id=row.id,
                    creator = new UserResponseType{
                       id=row.creator.id,
                       username=row.creator.username
                    },
                    description = row.description,
                    picture = row.picture,
                    location = row.location,
                });
        }
        return events;
    }

    [HttpPost]
    public EventResponseType Post([FromBody] EventRequestType eventRequest)
    {
        int userId = int.Parse(HttpContext.Request.Cookies["user_id"]);
        Event newEvent;
        using (var context = new AppContext())
        {
            User creator = context.Users.Where(user => user.id == userId).First();
            newEvent = new Event{
                creator=creator,
                description=eventRequest.description,
                picture=eventRequest.picture,
                location=eventRequest.location
            };
            context.Events.Add(newEvent);
            context.SaveChanges();
        }
        return new EventResponseType{
            id=newEvent.id,
            creator=new UserResponseType{
                id=newEvent.creator.id,
                username=newEvent.creator.username,
            },
            description=newEvent.description,
            picture=newEvent.picture,
            location=newEvent.location
        };
    }
}
```

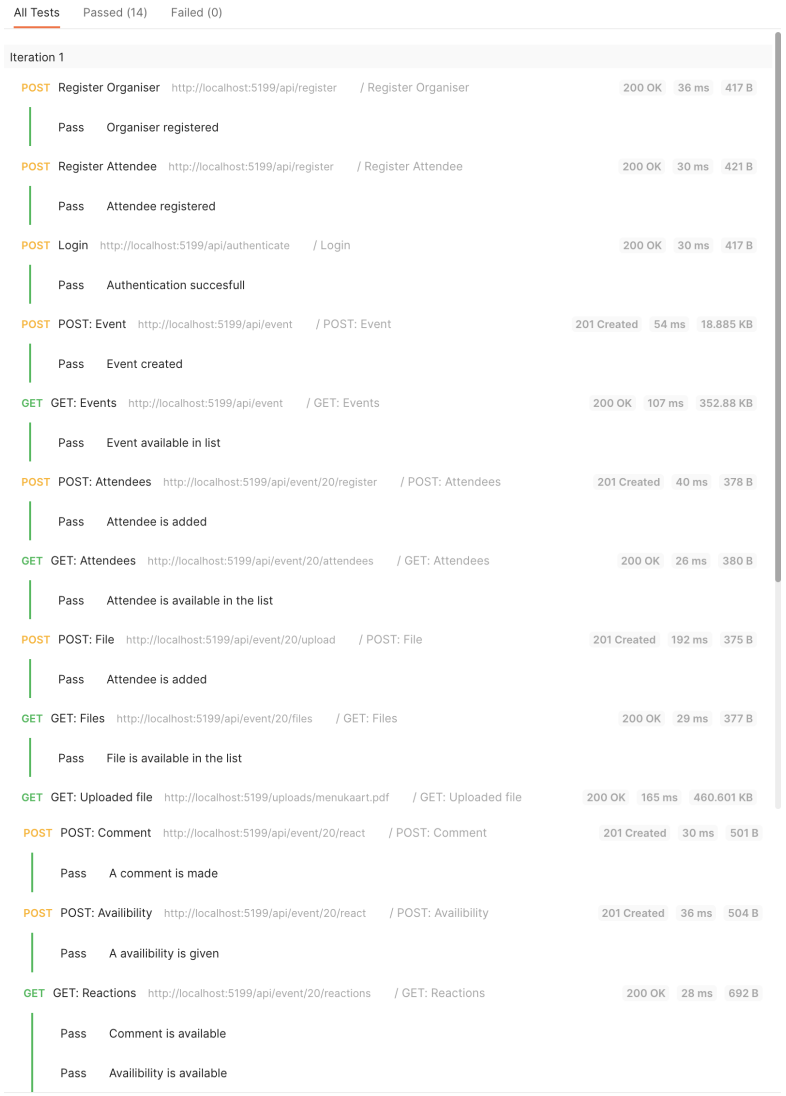
The code looks very similar to Node.js again, but this similarity is misleading. My initial feeling was that I'm working in a Node.js codebase, making it frustrating when the syntax was incorrect or when I have forgotten a semicolon. The C# The programming language is very strict on correct syntax. Besides this we also cannot get data from the database without opening the application context. Only then we can get the data using the entity classes. After this we need to convert all of the data to the exact format we need it in. This is pretty similar to the Node.js implementation. The difference however is that we need to use classes instead of custom data types. This is advised by Microsoft in their documentation, because it makes the code better understandable.

After having the entities, controller and classes for response types, we can run the application. To do this we need to use the dotnet watch run command. This will start the server and we can see our results on http://localhost:5199/

Most of the things I have learned on C# where by referring to the documentation. The documentation also felt elaborate and direct which I found useful. Because of this documentation I did not receive a lot of feedback on the code except for some variable naming.

D2: Testing software

Before I started development on the different programming languages, I assumed making tests in the programming language itself would be the best course of action. An acquaintance of mine brought to my attention that if I create the three proofs of concepts, I will need to make sure they work the same. This gave me the idea to make automated end to end tests which should succeed on every given prototype. I have done this by using a tool called Postman. This tool allows us to write application tests which we can execute on certain HTTP endpoints. After making all of the prototypes I have made these tests and I executed them on all proof of concepts. This failed initially because of some discrepancies. The tests eventually allowed me to correct the discrepancies in the proof of concepts and also proof the prototypes work how they should.



The testing language we have to use in Postman is Javascript. They are meant to test if all of the response data is available and automate a user flow. An example is that a user needs to register then we remember it's user id and use it to add that user to an event as an attendee.

```
pm.test("Attendee registered", function () {
    pm.response.to.have.jsonBody("id");
    pm.response.to.have.jsonBody("username");

    let json = pm.response.json();
    postman.setEnvironmentVariable("attendee_id", json["id"])
});


pm.test("Attendee is added", function () {
    pm.response.to.have.jsonBody("id");
    pm.response.to.have.jsonBody("user");

    let json = pm.response.json();
    pm.expect(json["user"]["id"]+"").to.eql(pm.environment.get("attendee_id"));
});
```