Luca Picovici
Software Engineering

Simple Proof of Concept Blog App

Service Oriented Architecture

1. Application

I have developed a simple proof of concept blog app that uses microservices internally. Its functionalities are creating posts, adding comments to the posts and moderating the comments based on a word, for example ‘orange’.

I have written multiple independent services that are deployed in a Kubernetes cluster.

Below is the UML diagram of the app.

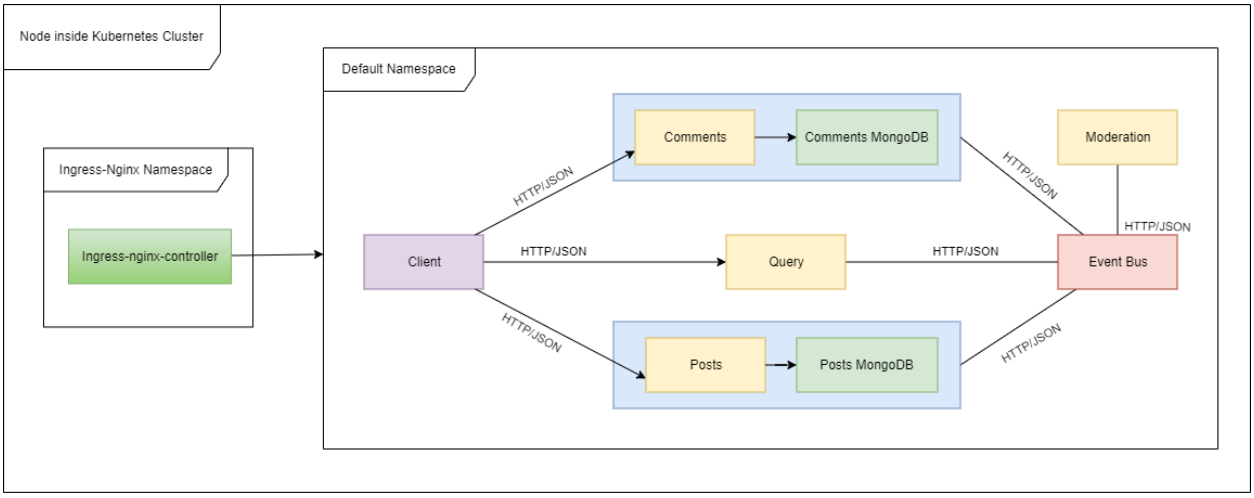

Fig. 1. UML diagram

The workflow can be explained like this – a request is made from the React frontend client in the browser to create a new post with the URL `https://blogapp.dev/api/posts` . Since the whole application is deployed to a Kubernetes cluster, the request will be passed to the Ingress Controller, which has the job to correctly route it to the Posts service. On the Posts endpoint, the request body is being validated and a post object is created and saved in MongoDB service. Finally, an event object is sent to the Event Bus service with type ‘PostCreated’ and the data. On the Event Bus endpoint, the event object is received and forwarded to all other services that are interested in the data.

When a comment is created, a request is made from the client to URL `https://blogapp.dev/api/comments/${postId}/comments` to create a new comment. The request reaches Comments service where the body is validated and a comment is created with the ‘default’ status attribute and saved in the corresponding Comments MongoDB service. Finally, as with

Posts, an event object is sent to the Event Bus service with type ‘CommentCreated’ and the data. Again, the event object is being forwarded to all other interested services.

When the Moderation service receives an event of type ‘CommentCreated’, it proceeds to validate the comment. In this case, it’s a simple verification if the content contains the word ‘orange’. If it does, the status is updated to ‘rejected’, otherwise ‘approved’. Finally, a ‘CommentModerated’ event type is being sent to the Event Bus. Now the Comment service receives this event and updates the corresponding comment. It then emits a ‘CommentUpdated’ event. If status is ‘rejected’, on the frontend it will appear ‘This comment has been rejected’, or ‘This comment is awaiting moderation’ in case the Moderation service is not working.

The Query service is also interested in these events because it assembles the post with its comments and stores everything in a data structure, ready for querying. The React client fetches the posts from the Query service with the help of useEffect function.

2. Microservices patterns

Some of the patterns used in developing the microservices architecture include Database for Service, for example there is a separate database service for Posts and Comments because they need to store the MongoDB model, whereas the other services don’t necessarily need a database for the purpose of this application. Query only stores the assembled posts in a JS object. Each database service has its own Deployment file to be used in Kubernetes.

Service component testing has also been used to isolate a service and test its functionalities using Jest and supertest.

For Server-side service discovery, Ingress Nginx Controller has been used because it receives all traffic in one endpoint and redirects the request to the corresponding service path, for example ‘/api/posts’. Code will be provided in the next section to explain a little more.

3. Code examples

The router below is for creating a new post at /api/posts using TypeScript and NodeJS.

```
router.post(
  '/api/posts',
  [
    body('title')
      .trim()
      .isLength({ min: 1, max: 100 })
      .withMessage('Title must be between 1 and 100 characters'),
  ],
  validateRequest,
  async (req: Request, res: Response) => {
    const { title } = req.body;

    const post = Post.build({ title });
```

```
    await post.save();

    await axios
      .post('http://event-bus-srv:3000/api/event-bus/events', {
        type: 'PostCreated',
        data: {
          id: post._id,
          title,
        },
      })
      .catch((err) => {
        console.log(err.message);
      });

    res.send(post).status(201);
  }
);
```

Testing the new post route with Jest and supertest

```
it('has a route handler listening to /api/posts for post requests', async () => {
  const response = await request(app).post('/api/posts').send({});

  expect(response.status).not.toEqual(404);
});
```

Post model in MongoDB

```
// An interface that describes the properties that are required
// to create a new Post
interface PostAttrs {
  title: string;
}

// An interface that describes the properties
// that a Post Model has
interface PostModel extends mongoose.Model<PostDoc> {
  build(attrs: PostAttrs): PostDoc;
}

// An interface that describes the properties
// that a Post Document has
interface PostDoc extends mongoose.Document {
  title: string;
```

```
  updatedAt: string;
}

const postSchema = new mongoose.Schema(
  {
    title: {
      type: String,
      required: true,
    },
  },
  {
    toJSON: {
      transform(doc, ret) {
        ret.id = ret._id;
        delete ret._id;
        delete ret.__v;
      },
    },
  }
);
postSchema.statics.build = (attrs: PostAttrs) => {
  return new Post(attrs);
};
const Post = mongoose.model<PostDoc, PostModel>('Post', postSchema);
```

Request validation custom error used when creating posts or comments. It is included and deployed among other errors and middlewares in a NPM package to prevent duplicating the files over all services where required.

```
export class RequestValidationError extends CustomError {
  statusCode = 400;

  constructor(public errors: ValidationError[]) {
    super('Invalid request parameters');

    // Extending a built-in class
    Object.setPrototypeOf(this, RequestValidationError.prototype);
  }

  serializeErrors() {
    return this.errors.map((err) => {
      return { message: err.msg, field: err.param };
    });
  }
}
```

4. Starting the application

To start the application, Docker, Kubernetes and Skaffold need to be installed. Skaffold monitors all the deployment files of all services and starts them. Once they are installed, run 'npm install' and in the root folder run 'skaffold dev'.